Black-Box Test Case Selection and Prioritization for Software Variants and Versions

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von

Remo Lachmann

geboren am 18. Oktober 1987

in Gera

Eingereicht am: 28.04.2017

Disputation am: 04.08.2017

1. Referentin: Prof. Dr.-Ing. Ina Schaefer

2. Referent: Ass.-Prof. PD Dr. Michael Felderer

2017

Abstract

Software testing is a fundamental task in software quality assurance. Especially when dealing with several product variants or software versions under test, testing everything for each variant and version is infeasible due to limited testing resources. To cope with increasing complexity both in time (i.e., versions) and space (i.e., variants) of software systems, new techniques have to be developed to focus on the most important system parts for testing. In the past, regression testing techniques such as test case selection and prioritization have emerged to tackle these issues for single-software systems. However, testing of variants and versions is still a challenging task, especially when no source code is available. Most existing regression testing techniques analyze the source code to identify changes which indicate test cases to be executed again, i.e., they are likely to reveal a failure. To this end, this thesis contributes different testing techniques for both, variants and versions, to allow more efficient and effective testing in difficult black-box scenarios by identifying important test cases to be re-executed. Four major contributions in software testing are made. (1) We propose a test case prioritization framework for software product lines based on delta-oriented test models to reduce the redundancy in testing between different product variants. The framework is flexible and allows to prioritize test cases for individual product variants. (2) We introduce a risk-based testing technique for software product lines. Our semi-automatic test case prioritization approach is able to compute risk values for test model elements and scales with large numbers of product variants. (3) For black-box software versions, we provide a novel test case selection technique based on genetic algorithms. In particular, seven different black-box selection objectives are defined, thus, we perform a multi-objective test case selection finding Pareto optimal test sets to reduce the testing effort. (4) We propose a novel test case prioritization technique based on supervised machine learning. It is able to imitate decisions made by experts based on training data available in system testing, such as natural language test case descriptions and black-box meta-data. All of these techniques have been evaluated using the Body Comfort System case study. For testing of software versions, we also assesses our testing techniques using an industrial system. Our evaluation results indicate that our black-box testing approaches for software variants and versions are able to successfully reduce testing effort compared to existing techniques.

Zusammenfassung

Testen ist eine fundamentale Aufgabe zur Qualitätssicherung von modernen Softwaresystemen. Mangels limitierter Ressourcen ist gerade das Testen von vielen Produktvarianten oder Versionen sehr herausfordernd und das wiederholte Ausführen aller Testfälle nicht wirtschaftlich. Um mit der Raum- (Varianten) und Zeitdimension (Versionen) in der Entwicklung umzugehen, wurden in der Vergangenheit verschiedene Testansätze entwickelt. Es existieren jedoch nach wie vor große Herausforderungen, welche es zu lösen gilt. Dies ist vor allem der Fall, wenn der Quellcode der getesteten Softwaresysteme unbekannt ist. Das Testen von Black-Box-Systemen erschwert die Identifikation von zu testenden Unterschieden zu vorher getesteten Varianten oder Versionen. In der Literatur finden sich bisher wenige Ansätze, welche versuchen diese Herausforderungen zu lösen. Daher werden in dieser Dissertation neue Ansätze entwickelt und vorgestellt, welche beim Black-Box Testen von Software-Varianten und -Versionen helfen, wichtige Testfälle zur erneuten Ausführung zu identifizieren. Dies erspart die Ausführung von Testfällen, welche weder neues Verhalten testen noch mit hoher Wahrscheinlichkeit neue Fehler zu finden. Insgesamt leistet diese Dissertation die folgenden vier wissenschaftlichen Beiträge: (1) Ein modell-basiertes Framework zur Definition von Testfallpriorisierungsfunktionen für variantenreiche Systeme. Das Framework ermöglicht eine flexible Priorisierung von Testfällen für individuelle Produktvarianten. (2) Einen risiko-basierten Testfallpriorisierungsansatz für variantenreiche Systeme. Das Verfahren ermöglicht eine semi-automatisierte Berechnung von Risikowerten für Elemente von Produktvarianten und skaliert mit großen Produktzahlen. (3) Ein multi-kriterielles Testfallselektionsverfahren für den Regressionstest von Black-Box Software-Versionen. Es werden verschiedene Black-Box Testkriterien aufgestellt und mittels eines genetischen Algorithmus optimiert um Pareto-optimale Testsets zu berechnen. (4) Ein Testfallpriorisierungsverfahren für Black-Box Regressionstests mit Hilfe von Machine Learning. Der verwendete Algorithmus ermöglicht eine Imitation von Entscheidungen von Testexperten um wichtige Testfälle zu identifizieren. Diese Ansätze wurden alle mit Hilfe von Fallstudien evaluiert. Die resultierenden Ergebnisse zeigen, dass die Ansätze die gewünschten Ziele erreichen und helfen, wichtige Testfälle effektiv zu identifizieren. Insgesamt wird der Testaufwand im Vergleich zu existierenden Techniken verringert.

Acknowledgements

In the years I was working on this thesis many people have supported me in different ways. I give my sincere thanks to the most influential persons in the following.

First and foremost, I thank my supervisor Ina Schaefer for giving me the chance to do my PhD at her institute. She has always given me great advice and provided me with the freedom to pursue different ideas and interests, which led to the diversified contributions of this thesis. I thank my reviewer Michael Felderer for taking the effort to review my PhD thesis, attending the defense and his advices for our paper.

I thank my colleagues at TU Braunschweig for creating a great working atmosphere, always having an open ear for problems and overall for a great time. In particular, I want to thank Sandro Schulze for guiding me in the difficult process of a writing successful papers. Another big thanks goes to Sascha Lity, who supported me in many fruitful discussions to shape the SPL testing concepts and gave me important feedback to improve my research. In addition, I thank Stephan Mennicke for his help with some of the formal definitions in this thesis.

Besides from my colleagues, I also received great support by many students which I supervised over the years. In particular, I thank Sabrina Lischke and Simon Beddig for providing important preliminary work for contributions in testing of software variants. Furthermore, I thank Manual Nieke for his support as student researcher and for implementing the approaches to test black-box software versions.

I thank my mother Silke for always supporting my decisions and never doubting my abilities. She has always enabled me to think outside the box and to be creative. My father Peter has awaken my wish to see the world, which became one reason to enjoy my PhD time even more. I thank my grandmother Irene for her support in my education and presenting abilities.

Many thanks go to my friends from Braunschweig, Leizpig, Würzburg and Gera. They supported my social life and made these years enjoyable and they helped me to take my mind off from work.

Finally, I thank my beloved wife Karina, who has accompanied me trough all steps of this thesis and beyond. In bolstering me and sharing empathy for my problems, she has given me the strength to strive for this achievement. I am very happy to share this success with her!

Contents

Part I

Preliminaries

1 Introduction

Modern software systems are often developed in software engineering processes, with different aspects of development dedicated to distinct phases. Examples of software development processes are the Waterfall model, V-model or Scrum [Som10]. Within these processes, software testing is a fundamental task to ensure software quality according to the specification of the system. Testing consumes a large amount of time and resources due to the complexity of modern systems. A proficient testing process increases the trust in the completed product. Especially in safety-critical software systems, such as automotive electronic control units, it is of utmost importance to reduce the failure rate and, thus, the involved risk [LS14]. While testing has been a research focus for decades, there are still problems to be solved. In this chapter, we motivate the current problems in testing considered in this thesis and give an overview of our contributions. Finally, we briefly present the structure of this thesis.

1.1 Problem Statement

Software Variants and Versions. Modern software systems are continuously evolved to fix bugs or introduce new features, which leads to new software versions [Leh80]. Similarly, different functionality might be provided for different product variants of the same software system [PBvdL05]. Here, the customer is able to select a desired set of functionalities to personalize the product. To illustrate the two dimensions, i.e., versions (over time) and variants (product space), we show an example in Figure 1.1. The figure depicts two different versions of the Windows operating system[1]: Windows 8 and its successor Windows 8.1. Considering the product space dimension, each version has been published in different *variants*, e.g., Windows and Windows Pro as shown in Figure 1.1.

Current Challenges in Software Testing. While software variants and versions are prominent in modern software systems, testing them is still a difficult task due to restricted testing resources. Especially large numbers of variants and fast development cycles for new software versions increase the complexity of testing, making a full test of each variant and version infeasible [Eng10, MMCDA14].

[1] Windows is developed by Microsoft Corporation, official website: `https://www.microsoft.com/en-us/windows/`, date: March 20 2017

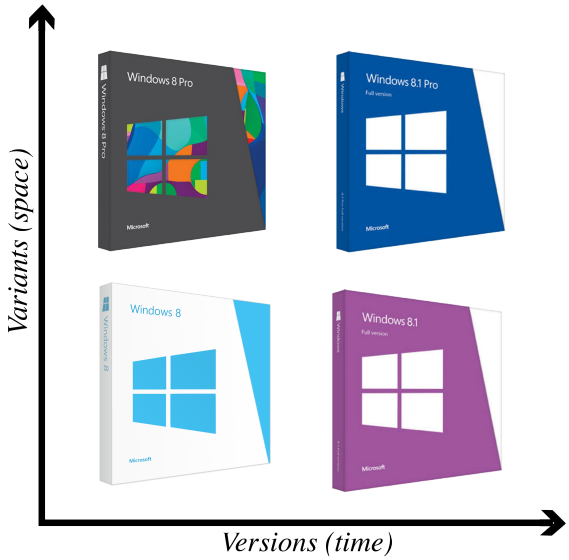

Figure 1.1: Example for Development of Software Variants and Versions

For both dimensions, test cases have to be repeatedly executed to ensure that a new variant or version corresponds to its specification. Several testing techniques have been proposed in the past to reduce testing effort and guide testers to identify important test cases. For testing of software versions, the most prominent examples to reduce testing effort are *test case selection* and *test case prioritization* techniques [YH07b]. However, these regression testing techniques usually assume to have source-code knowledge [RH94, YH07a, EYHB15, MHD15] or are model-based [CPS02, BLC13, RBT13]. Unfortunately, complex software systems are often developed as component-based systems, i.e., they are a composite of different, sometimes independently developed, software components. While components increase software reusability and support parallel development, they introduce new challenges in testing as the source code of such components is not always available [Wey98], for instance, in case that components are developed by external companies. Only few techniques deal with black-box meta-data, such as test case history [FKAP09, ERL11, QNXZ07]. There is no technique which makes use of natural language artifacts or incorporates many black-box objectives at once for test optimization. In addition, the complexity of variant-rich software systems has not been explored for testing individual variants. Current techniques for testing of different software variants mostly focus on the selection [JHF12, LOGS11] or prioritization [AHTM+14, DPC+13, PSS+16, AHLL+17] of product variants or test case generation [UKB10, VBM15]. However, only few techniques make contributions to improve testing of individual variants [AWSE16, LLL+14, WAG15].

The identification of important test cases for *black-box* software variants and versions is difficult as code-level changes are not directly observable and analyzable. Thus, new techniques are required to deal with the complexity of testing of black-box software variants and versions.

Black-Box Testing of Software Variants. One trend in modern software systems is the deviation from single-software systems to variant-rich software systems, such as *software product lines* (SPL). SPLs allow for mass customization of product variants to fulfill personalized needs and functionalities, e.g., by providing more complex variants with additional functionality for a higher price [PBvdL05]. One prominent example for these types of systems can be found in the automotive domain, where cars can be highly customized, e.g., by providing different types of driver assistance systems. However, while variant-rich systems gain popularity, they introduce new challenges for engineering and, especially, testing. One major challenge is the potentially high number of derivable product variants. Another problem is the high degree of redundancy between product variants, as they share common functionality. This increases the effort in testing as there is high chance that test cases are executed redundantly. Thus, identifying important test cases for individual product variants is a crucial but difficult task.

While techniques have been proposed to improve testing of variant-rich software systems in the past [Eng10], we argue that there is a distinct lack of techniques to identify important test cases for particular product variants. Currently, most testing techniques focus on the derivation of subsets of product variants to be tested, but do not investigate how to test each of these variants. This is a major drawback as each product variant has the complexity of a single-software system and, thus, a full test of each derived product variant is not feasible.

We propose novel approaches to deal with the issues of prioritizing test cases for each product variant under test according to different criteria. Providing test case prioritization allows testing of a product variant to continue until resources are exhausted, while ensuring that the most important test cases have been executed. In particular, we provide a flexible framework for black-box testing of variant-rich systems to identify changes between variants using test models. These changes indicate parts in product variants, which are likely to produce new failures and, thus, should be prioritized in testing. In addition, we provide a testing approach, which semi-automatically identifies risky parts of the system. Test cases are prioritized according to the riskiness of their covered software parts. Both techniques are applicable for large scale variant-rich systems and are able to provide test case priorities for individual product variants, while requiring only low manual effort. Our SPL related techniques do not require any code access and improve the coverage of changes between product variants. Instead, we rely on test models as provided in model-based testing [UL07].

Black-Box Testing of Software Versions. While a high degree of product variants introduces new challenges, testing of single-software systems is also a difficult task due to continuous time-restricted retesting of new software versions. In this context, this thesis contributes to black-box *regression testing* of software versions. Regression testing focuses on retesting of previously tested software parts, which might have been influenced by changes performed in a new software version [RH94]. While many regression testing techniques exist [YH07b], we argue that there is a distinct lack of sufficient regression testing techniques when dealing with black-box systems. Without source code access, the identification of potentially influenced system parts is a non-trivial task, which depends on expert knowledge and repeated manual assessments. This makes a reliable identification of important regression test cases difficult and expensive, especially for a large number of test cases. To this end, we provide two different approaches to select and prioritize test cases in regression testing without analyzing source code.

First, we use genetic algorithms to optimize the selection of test cases according to several objectives at once, resulting in a multi-objective optimization technique. Our objectives are based on meta-data available in system testing, such as preferring prefer test cases which have revealed failures in the past. Consequently, we identify Pareto-optimal solutions to the black-box test case selection problem. Evaluation results indicate that our test case selection technique is able to reduce testing effort.

In a second contribution to black-box testing of software versions, we prioritize test cases based on machine learning. Here, we focus on natural language test cases and analyze their descriptions and meta-data to learn a ranked classification model, which allows us to identify the priority of test cases. The approach aims to emulate test experts, which provide training data (i.e., important and unimportant test cases) to train the machine learning algorithm. To our knowledge, this is the first work in which natural language test cases are prioritized based on machine learning techniques. Our results show that this approach is able to find failures early in real-life data. In addition, it outperforms human test experts in terms of failure finding rate.

1.2 Contributions

This work aims to improve black-box testing for both, software variants and versions. Consequently, the contributions are twofold: The first two contributions improve black-box testing of software variants while the latter two improve black-box testing of versions. In summary, the four main contributions of this thesis are as follows:

1. A model-based delta-oriented test case prioritization framework for black-box testing of software variants [LLL+15, LLAH+16, LLS+17]. The framework is

designed to be extensible and flexible. The framework's concepts are tailored for reuse of test cases and allows the user to prioritize test cases for individual product variants. In particular, the used techniques are similar to regression testing techniques of single-software systems, but applied to the variability in space dimension.

2. A risk-based test case prioritization approach for black-box testing of software variants [LBL+17]. Our test case prioritization technique computes risk values for generic test models in a semi-automatic fashion. This reduces the testing effort significantly compared to traditional risk-based testing approaches applied to single-software systems.

3. A multi-objective regression test case selection approach for black-box testing of software versions [LFN+17]. We propose seven black-box test objectives to be optimized. Genetic algorithms are used to find Pareto-optimal test sets. Results show good results in terms of precision and recall.

4. A test case prioritization for testing of software versions [LSN+16]. We use supervised machine learning to emulate decisions and experiences made by test experts to improve black-box testing based on natural language test cases. Results indicate that our test case prioritization technique is indeed able to improve regression testing, even when compared to a human test expert.

We use the Body Comfort System case study to evaluate our black-box testing techniques [LLLS12]. For our software version testing, we additionally performed evaluations with real-life data from the automotive industry.

1.3 Outline

Due to the nature of our contributions, we split this thesis into four parts. Part I consists of three chapters, of which this introduction chapter is the first. We provide important background and terminology for this thesis in Chapter 2. In Chapter 3, we explain the foundations for the contributions of this thesis. In particular, we provide formal definitions of used artifacts and describes the Body Comfort System (BCS) case study, which is used to evaluate our contributions.

Part II comprises our contributions for black-box testing of software variants. It contains two chapters. Chapter 4 introduces our model-based test case prioritization framework for SPLs (Contribution 1). The framework supports testing of product variants in large SPLs as it provides concepts to prioritize test cases for each product variant under tests separately. Chapter 5 provides a risk-based test

case prioritization approach for SPLs, which computes risk values for test model elements semi-automatically (Contribution 2).

Part III of this thesis is split into three chapters. Chapter 6 presents our multi-objective regression test case selection technique for system-level testing without source code access (Contribution 3). Here, we use genetic algorithms to find Pareto-optimal test sets which optimize several objectives at once. Chapter 7 concludes our black-box testing technique for software versions with a test case prioritization approach, based on machine learning (Contribution 4). We summarize this thesis in Chapter 8, where we discuss the obtained results. In addition, we present potential future work to extend our black-box testing techniques or show its general applicability. We present and explain related work for each contribution individually, showing the novelty of our black-box testing approaches.

Part IV concludes this thesis with an appendix. It further contains an index and a list of abbreviations.

2 Background

To lay the foundation of the proposed contributions in this thesis, we explain necessary background in this chapter. First, the basic concepts and ideas of software testing are explained. This includes the V-model, black-box testing and model-based testing (MBT). Next, regression testing is explained. In particular, the concepts of test case selection and prioritization are introduced. Afterwards, software product lines (SPL) are introduced, including the SPL engineering process, SPL modeling techniques and delta-oriented modeling.

2.1 Software Testing

Modern software systems are developed according to software engineering principles, containing different aspects to be fulfilled to produce software systems of high quality [Som10]. One popular software development process is the *V-Model*, consisting of different engineering and testing phases to guide and support software development. The contributions of this thesis are aligned within two testing phases of the V-Model. Thus, we explain the V-Model in more detail in the following. In this context, the applied testing terminology for the remainder of this thesis is introduced. Afterwards, we give some background on black-box testing which is a particular domain of testing techniques dealing with lack of source code and the underlying assumption for all our proposed contributions. To conclude this section, we explain necessary background on regression testing, which is a fundamental aspect of this thesis.

2.1.1 Testing in the V-Model

A wide variety of software development processes [Som10] have been developed in the past, e.g., the *Waterfall model*, the *V-model* or agile techniques like *Scrum* [Sch04]. In context of this thesis, we focus on the V-model, which is a widely accepted and applied software development process. Different version of the V-Model exist, with adaptations being made for different domains (e.g., automotive software engineering) or project specific needs. We describe the V-model as introduced by the ISTQB[1] in

[1]The *International Software Testing Qualifications Board* creates software quality assurance certifications for software testers. Link: `http://www.istqb.org`, date of visit: August 17th 2016

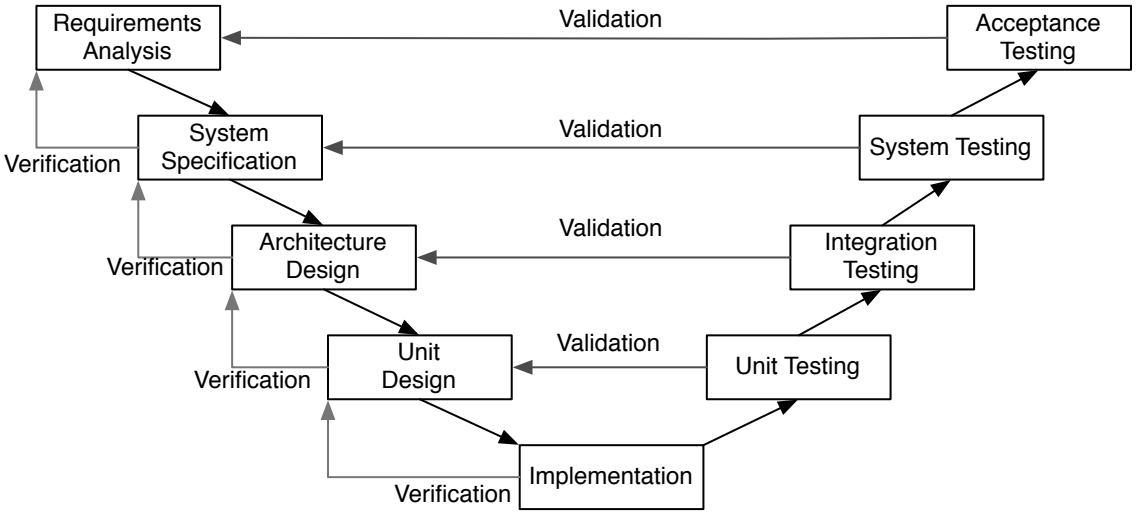

Figure 2.1: The V-Model Software Engineering Process (modeled after [SLS11])

the context of this thesis as this version is taught to certified testers [SLS11]. The nine distinct phases of the V-model and their relationship are shown in Figure 2.1.

Development Phases. The left side of the *V* comprises five *development phases*, which lead to the implementation of the software system. In the V-model, each phase is the foundation for the next phase, producing the artifacts needed for the next phase to commence. The development is initiated with a *requirements analysis*, which is performed in unison with the customer and stakeholders until an informal specification of the system is derived. Based on this informal specification, the *system specification* is designed, which is used as definition of the system. Using the system specification, an *architectural design* of the software system is formulated. It describes the communication and interaction of the different components of the system via interface specifications, i.e., it describes the system architecture. Finally, all components are individually designed (*unit design*), such that they can be implemented in isolation. Thus, they comprise atomic functionalities and provide or require the interfaces specified in the architecture design. Based on these designs, the software is *implemented* in the last step of the left side of the V-model.

Testing Phases. The four *testing phases* are aligned on the right side of the V. Each conceptual development phase has a counterpart on the testing side, i.e., the design of the system provides the *specification* for the corresponding testing phase on different development levels. In each testing phase, the specification is used to derive a set of *test cases* $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$ for the *system under test* (SUT). Each test case covers a certain aspect of the SUT and ensures its conformity to the specification. In

unit testing, test cases assess each component or unit in isolation on a very technical level. For *integration testing*, test cases validate that the architectural design has been implemented as specified, i.e., that the communication of components is correct and that the interfaces have been implemented properly. In *system testing*, test cases correspond to the requirements of the customer or stakeholder, i.e., the system is tested as a whole according to specified use cases. *Acceptance testing* directly involves the customer and is tested on site on customer hardware. The later failures are found in the testing process, the more expensive they are to fix [Som10]. Thus, each underlying testing phase should be executed before the next.

In testing, testers aim to find *faults* in the code, i.e., faulty statements. These are most likely found when testing directly on code-level, e.g., in unit testing. A fault will most likely lead to one or more user-observable *failures*, e.g., a crash of the system or wrong computations. In black-box testing, test cases reveal failures instead of faults as only the result is observable, but not the cause. The mapping of a failure to the underlying fault(s) is part of the debugging process.

This thesis mainly contributes in the integration and system testing phases. Thus, we explain these two testing phases in more detail in the following.

Integration Testing. The architecture design of a software system represents its component-based structure and its inter-component communication [Som10]. Each *component* has a specified interface which defines its interactions with other components, i.e., which *signals* are received and sent via *connectors*. Thus, integration testing aims to validate if the internal communication has been properly implemented using the architecture design as specification.

Due to the complexity of modern software systems they usually comprise several components. For testing, it should be avoided to integrate all components at once as this might lead to *failure masking*, i.e., if more than two components are involved in a erroneous communication, the identification of the actual problem is difficult [SLS11]. Thus, integration testing should always start with a minimal subset of components, ideally a pair of components. Further components can be added in a step-wise fashion to test more complex subsystems. As subsystems might require different components, which are not available, *drivers* and *stubs* have to be used to simulate missing components and enable a controlled communication between the components under test [Som10, SLS11].

Two basic integration strategies are *top-down-* and *bottom-up*-integration [SLS11]. They define the ordering in which additional components are integrated into the subsystem under test. In case of top-down, the integration testing process starts with the component(s) which call other components and are never called by other components. Integration then continues with the components called by the integrated components until, at last, the components are integrated which do not call any other components. The bottom-up integration works similarly, but in reverse.

System Testing. Once all components are integrated and tested, *system testing* commences. Here, the overall system is the test subject. A system is specified by a set of requirements $\mathcal{REQ} = \{req_1, \ldots, req_n\}$, which are defined stakeholders involved. Requirements are either *functional* or *non-functional*, i.e., they either specify what functionalities shall be enabled or how the system shall behave [Som10]. As requirements are derived by the customer, they are usually defined in natural language to support the comprehensibility of the SUT.

To assess if a system performs according to its specification, a set of *system test cases* is defined. *Traceability* between requirements and test cases is important to assess the quality of the system. In other words, it is important to know which test case covers which requirement to be able to measure certain quality-related metrics. One basic assumption for system testing is that each requirement is covered by at least one test case, i.e., *requirements coverage* [WRHM06] is required to ensure a certain quality.

In modern software systems, many system level test cases are executed manually as they represent use cases performed by end users [Sne07]. In addition, system testing often involves testing of graphical user interfaces to ensure certain user interaction scenarios. Thus, test automation is more difficult compared to, e.g., unit testing. Manual testing leads to a high effort in testing, especially with restricted testing resources, which is on of the challenges tackled in this thesis.

2.1.2 Black-Box Testing

White-Box vs. Black-Box Testing. Software testing is concerned with the creation, execution and evaluation of test cases [AO08]. Testing requires as system specification to derive *test oracles*, i.e., to know the expected results of each test case, which enables the tester to determine whether the system behaves correctly [RAO92]. To design test cases, literature distinguishes two different testing domains: *white-box* and *black-box* testing. In white-box testing, the source code is available for test case design, e.g., in unit testing.

White-box testing allows to apply code-based coverage criteria, which support the test case design process. Sample white-box test coverage criteria are *statement-*, *transition-* or *path*-coverage [SLS11]. Statement coverage demands that each code statement is executed at least once. Transition coverage extends this notion and forces the execution of each transition, i.e., the control flow of the software is tested. Path coverage is a theoretical coverage criterion as it demands each execution path to be executed at least once, which is infeasible when dealing with loops.

However, source code might not always be available or too complex to analyze for white-box testing. In this case, the software is tested using *black-box* testing techniques, i.e., testing is solely based on the system specification. The specifica-

tion might exist in form of requirements (cf. Chapter 2.1.1) or test models (cf. Chapter 2.1.3).

Component-Based Software. One prominent type of systems for which black-box testing is often inevitable are component-based software systems [Wey98, HLS99]. These types of systems consist of a set of components which communicate with each other. Off-the-shelf components can be assembled to form well-defined software architectures, which reduces the development cost and avoids the development from scratch [CLWK00]. One example for component-based software systems are automotive applications, where software is deployed to *electronic control units* (ECU). Each ECU can either contain one component or components might be distributed over several ECUs [Bro06]. However, each ECU needs to be implemented and connected correctly to provide the desired functionality, e.g., autonomous driving of the vehicle. The complexity of different interacting components makes testing of component-based software systems difficult [HLS99]. Among others, the following reasons are responsible for the black-box testing nature of component-based software [Wey98]:

- **External Developers:** Especially in complex software systems, not every component is created by the same developer. Instead, external suppliers provide necessary functionalities for certain components of the system [Wey98]. Due to legal restrictions, the source code of these components might not be available and, thus, is provided as a black-box where only information about public interfaces is known [WCO03]. While a white-box unit test is usually provided by the supplier, integration of these components requires black-box testing techniques to ensure their correct communication.

- **Reuse of Components in Different Projects:** In most cases, modern software systems are not developed from scratch but reuse, adapt or extend existing code to fulfill new needs [Leh80, HvKS08]. Similar to external developers, the reused source code might not be directly available but only in binary form. The same issues rises with external closed-source libraries.

- **Large Quantities of Code:** Complex systems consist of up to millions of lines of code, which are hard to analyze and to understand. No single person knows all details about the code, which makes a white-box analysis of complex and component-based software systems difficult [Wey98].

Black-Box Testing. Black-box techniques are important when dealing with modern, complex software systems. Black-box testing is prominent in integration, system and acceptance testing as it assumes knowledge about external input and output interface of the SUT (cf. Figure 2.2). Hence, the *point of control* and *point*

Figure 2.2: Points of Control and Observation in Black-Box Testing

of observation are outside of the systems interior structure, i.e., no knowledge about source code is available and only observable outputs define the result of a test case.

Existing black-box testing techniques often focus on the test case creation process, e.g., equivalence classes, boundary value analysis or decision tables [LV04]. However, the execution of test cases is not trivial, especially in the testing of different software versions where only a subset of test cases can be executed again due to resource limitations. This *regression testing* (cf. Chapter 2.2) of different software versions is important, but there is a lack of black-box regression testing techniques [ERS10].

2.1.3 Model-Based Testing

While black-box testing is a common task in software engineering, it is still difficult to deal with a lack of source code availability. Especially in terms of test case creation and complexity of the specification difficulties arise when requirements are defined in natural language. To this end, *model-based testing* (MBT) has been introduced to cope with black-box systems and introduce new test end criteria and test case generation techniques [UL07]. MBT always requires the existence of a *test model*, which is a formal representation of the system specification. In the context of this thesis, we use *structural* test models, i.e., *architecture models* and *behavioral* test models, i.e., *state machines*.

Architecture Test Models. Modern software systems are usually defined in a *component-based* fashion [CLWK00], i.e., a system is described as a set of components $C = \{c_1, \ldots, c_n\}$. Components are defined in the context of a *software architecture* [Bas07]. They interact with each other via their provided *interface*. The foundation of the model-based development methodology is a well-defined *software architecture model*, which resembles the specification of a system's structure.

In context of this thesis, an *architecture model* contains a finite non-empty set of *components* $C = \{c_1, \ldots, c_n\}$, a finite non-empty set of *signals* $\Pi = \{\pi_1, \ldots, \pi_k\}$ and a *connector relation* $CON \subseteq C \times \Pi \times C$. Thus, we define an architecture model as triple $arc = (C, CON, \Pi)$. The *interface* of a component $c \in C$ is defined as the non-empty set of *incoming connectors* $I_c \subset CON$ and *outgoing connectors* $O_c \subset CON$. Using hierarchical layers, components might contain other components and

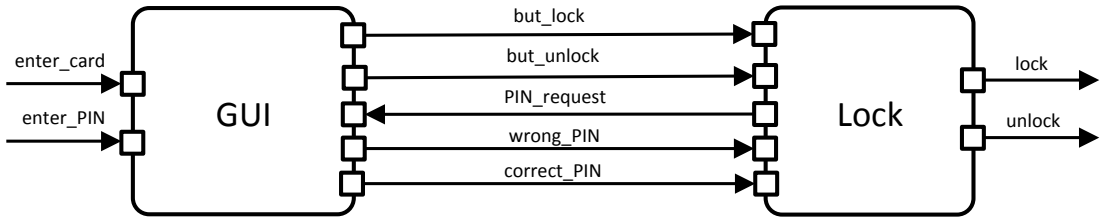

Figure 2.3: Sample Architecture Model for Locking Mechanism

connectors themselves, allowing for a more fine-granular description of the system structure. We require that each component has a connector to at least one other component and that the architecture is connected. A signal has a certain *type*, e.g., *boolean* or *integer*, and might be sent by more than one connector. Each connector transmits exactly one signal between exactly two components. Connectors are connected to either a in- or out-*port* of a component. We abstract from the notion of ports in the context of this thesis and assume that each connector is linked to the correct port of a component.

Architecture models can be either defined in a graphical fashion (cf. Figure 2.3), where a component is depicted as a box or node and connectors are depicted as arrows or directed edges. Another possibility are *architecture description languages*, which are a textual representation of the architecture [MT00]. As graphical representations are easier to grasp we provide graphical examples in the course of this thesis. The internal structure of components is not depicted in the architectural models as it is not of relevance on this level of abstraction.

In terms of testing, software architectures might be used as *test models* to specify the inter-component communication of a software system. Thus, they lay the foundation for integration testing (cf. Section 2.1.1). In context of this thesis, we use architecture test models to identify important test cases for integration testing of a certain variant of the system.

Example 2.1: Software Architecture

An example for a graphical architecture model is shown in Figure 2.3. This particular architecture comprises only two components, GUI and Lock. Assume, that the system describes a locking mechanism restricted by entering a PIN. The GUI component receives input from the *environment*, e.g., via buttons not represented. Then, the Lock component receives the input and if an entered PIN was correct or not. These signals are transferred via connectors, shown as arrows, e.g., (GUI, but_lock, Lock).

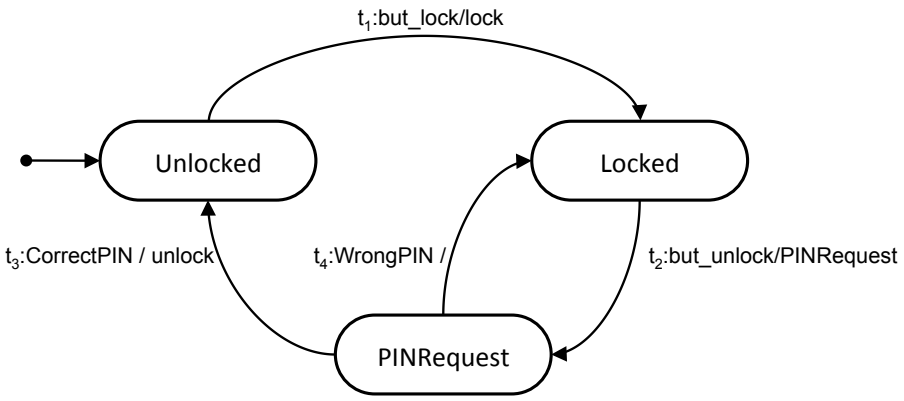

Figure 2.4: Sample State Machine for Locking Mechanism

State Machine Test Models. State machines are finite automata, which describe the behavior of a system or component [UL07], i.e., in which state a component is in and how it reacts to incoming events. One popular type of state machine are *state charts* [Har87], which are also part of the Unified Modeling Language (UML) [OMG15]. State charts support, for instance, parallel states, different hierarchy layers and guarded behavior. In context of this thesis, behavioral test models are used to represent the behavior of a component.

Let $SM = \{sm_1, \ldots, sm_i\}$ be a finite non-empty set of state machines. We define a *state machine* as 4-tuple $sm = (S, T, L, \Sigma)$ with a finite non-empty set of *states*, a finite nonempty set of *events* $\Sigma = \{\sigma_1 \ldots, \sigma_k\}$, which is divided into distinct sets of input event Σ_I, output events Σ_O and *internal* events Σ_τ, i.e., $\Sigma = \Sigma_I \cup \Sigma_O \cup \Sigma_\tau$ holds and a *transition relation* $T \subseteq S \times L \times S$ over the set of transition labels $L = (\Sigma_I \cup \Sigma_\tau) \times \mathcal{P}(\Sigma_O \cup \Sigma_\tau)$. When initiated, a state machine is in its initial state, usually notated by an arrow who originates in a black dot (cf. Figure 2.4 with initial state *Unlocked*). A transition $t \in T$ connects two states and defines how events are transferred between states to model the behavior of the system.

As state machines describe the behavior of a component, each input event $\sigma \in \Sigma_I$ correspond to a *signal* received by the component via its interface as incoming signal. Analogue, the set of output events Σ_O corresponds to outgoing signals on architecture level. We assume, that the mapping between events and signals is enabled via their names, i.e., they share identical names.

In contrast to input and output events, internal events Σ_τ do not correspond to architectural signals and are, thus, only visible within the state machine. Internal events enable control mechanisms required to describe the behavior of the component. While transitions are triggered by exactly one input event $\sigma_i \in \Sigma_I \cup \Sigma_\tau$, they may also *broadcast* a set of events $\Sigma_B \subseteq \Sigma_O \cup \Sigma_\tau$. The transition label syntax of

a transition $t \in T$ is defined as $t : \sigma_i / \Sigma_B$ (cf. Figure 2.4). Each state machine also has an *initial state* $s_{init} \in S$, which is entered when the machine is started. While parallel automata and hierarchy layers are part of state machines, we abstract from this to present a simple state machine definition that sufficiently serves the purpose of the testing concepts for software variants provided in this thesis. Furthermore, we focus on reactive systems, i.e, we assume that a state machine does not have particular *final state*, which brings the system to a halt.

Example 2.2: State Machine for Locking Mechanism

A sample state machine is shown in Figure 2.4. It represents the behavior of the locking component in Example 2.1. It is modeled in three states $S = \{\mathsf{Unlocked}, \mathsf{Locked}, \mathsf{PINRequest}\}$. The initial state $s_{init} = \mathsf{Unlocked}$ defines that the system is unlocked when started. The three states are connected via a total of four transitions $T = \{\mathsf{t_1}, \mathsf{t_2}, \mathsf{t_3}, \mathsf{t_4}\}$. For example, transition $\mathsf{t_1}$ requires the input event but_lock in order to be triggered, in which case it broadcasts the output event $\Sigma_B = \{\mathsf{lock}\}$. Assume, that these two events correspond to the component's interface shown in Figure 2.3 as the trigger might resemble a pressed button received as incoming signal and that the signal lock is send to other components in the system. In contrast, the events WrongPIN and CorrectPIN shall be internal τ-events, which are only visible within the state machine and used to control the inner workings of the state machine. For instance, the WrongPIN event does not lead to any changes in the internal state due to the still locked status.

2.2 Regression Testing

Reasons for Regression Testing. Testing is an ongoing process that continues even after the software has been released. Different reasons exist for ongoing testing processes [Leh80, LW89]:

- **New features are integrated:** Modern software systems are constantly updated, providing new functionality. Prominent examples are operating systems or firmware. Customers demand new features, which are build on top of existing software systems and are integrated after the initial release.
- **Faults are fixed:** Complete software testing is impossible [AO08]. Hence, testing is based on heuristics and test end criteria to find most of the existing faults. However, a released software system often still contains faults, which are noticeable by the customer or hinder the system's functionality. Thus, bug

fixes are provided by the developers via an update function, which alters the original software system to resolve existing failures.

- **Software is adapted:** Software systems might be adapted to many different hardware platforms or devices, e.g., new mobile devices. An adaption of the original software is necessary to ensure the correct functionality.

Regression Testing Techniques. Consequently, a change of the software requires testing to ensure that the system still fulfills its specification. In this regard, *regression testing* [LW89] focuses on already tested parts of an original software version and tries to ensure the correctness of these parts in new versions of the same application. One way to apply regression testing is a *retest-all* approach [YH07b]. Retest-all requires that out of all test cases $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$ those test cases, which are applicable to the current program version, are executed again. Of course, this testing technique has a major disadvantage: If the number of test cases is large, a lot of resources are put into redundant testing, which reduces testing efficiency. In addition, retest-all is usually infeasible as the effort of executing all test cases might be larger than available testing resources, i.e., not every applicable test case can be executed again for every version. This is an issue especially in agile release schedules, where regular updates are released in short time spans and testing has to be performed in an agile fashion [CG09]. Another issue arises if test cases are to be executed manually, e.g., in system testing (cf.Chapter 2.1.1), which reduces the testing throughput drastically and forces the tester to focus on subsets of all test cases [Sne07].

Example 2.3: Identifying Regression Test Cases

Assume that the software architecture of a software has changed from version P to version P' as shown in Figure 2.5. Each version comprises three components, A, B and C. In a tested version P, three connectors ensure the communication of the system. Their sent signals are not of relevance for this example. Now, an update of the software leads to version P' as shown in the right-hand side of Figure 2.5. As the example indicates, a new connector has been added between A and C. The tester has to decide what parts of the system are to be retested for regression testing. The *retest-all* regression criterion requires that the complete system is retested. Other techniques might focus on differences between product versions, i.e., in the example only the communication between A and C might be retested. However, the change might influence component A or C in an unpredicted way, leading to complications in the communication with B. This brief example illustrates the difficulties in finding sufficient regression test cases.

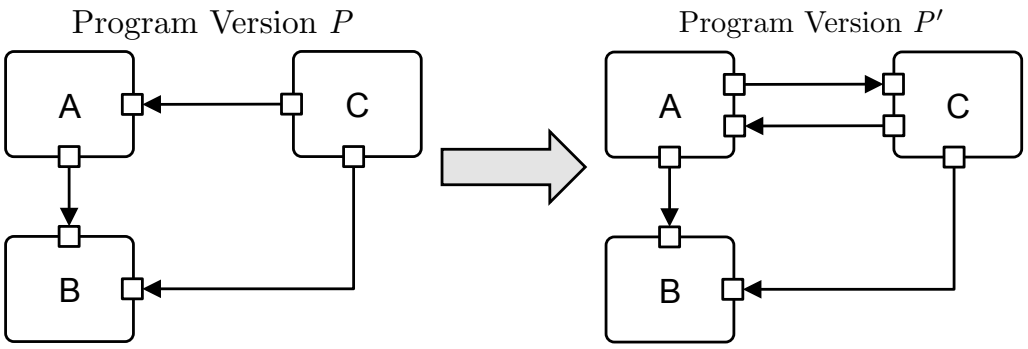

Figure 2.5: Example Changes between two Software Versions

To cope with limited resources in regression testing, i.e., budget, time or personnel, different regression testing strategies have been developed in the past. In particular, the three main regression testing approaches are *test case prioritization*, *test case selection* and *test suite minimization* [YH07b]. This thesis contributes test case selection and prioritization techniques in black-box testing.

2.2.1 Test Case Selection

Problem Definition. When dealing with a large set of test cases $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$, the *selection* and execution of a subset of test cases is a popular approach to reduce the testing effort [ERS10]. Thus, testing effort is reduced compared to a retest-all approach. According to Rothermel and Harrold [RH96], the test case selection problem is defined as follows:

Definition 2.1: Test Case Selection Problem

Let pr be a program, pr' a modified version of pr and $\mathcal{TC}$ a set of test cases.

Problem: Find a test set $TS' \subseteq \mathcal{TC}$ with which to test pr'.

Safe Test Case Selection. To solve the test case selection problem, a wide range of techniques has been proposed in the past [GHK+01, CPS02, YH07b, ERS10]. One approach is to focus on *safe regression test case selection* [RH97]. It avoids the removal of test cases which find a fault in the system, which is not revealed by any other test case. To attain this knowledge, source code availability is the foundation for most existing techniques. Source code can be used to construct control flow graphs and analyze which parts of the system are executed by what test case [RH97]. Modified parts of the software are then to be covered by a subset of regression test cases, to ensure that potential faults can still be revealed.

Test Case Categorization. In the context of test case selection, four test case *categories* are distinguished [LW89]:

New: When adding new functionality to a system, new requirements are specified. These requirements lead to a need for a set of *new test cases* $TC_{new} \subseteq \mathcal{TC}$. A new test case is usually of high priority as it has never been executed before and, thus, its execution should be mandatory for the current SUT.

Obsolete: Previously implemented functionality might be removed from a new version. This set of *obsolete test cases* $TC_{obsolete} \subseteq \mathcal{TC}$ is not part of the regression testing process for the current SUT as the test cases are no longer executable.

Reusable: A certain set of test cases has been executed for at least one previous version and is also applicable for the current version of the SUT. This set of *reusable test cases* $TC_{reuse} \subseteq \mathcal{TC}$ is the foundation for regression testing. While each test case of this set is potentially executable, there might be too many reusable test cases to execute all of them.

Retestable: To improve the efficiency of regression testing, a subset of *retestable test cases* $TC_{retest} \subset TC_{reuse}$ is selected from the reusable test cases. This set of test cases is to be executed for the current SUT and should be based on sophisticated regression test case selection techniques to ensure a high fault finding probability, i.e., the set should only contain effective test cases which reveal new faults.

Open Challenges in Test Case Selection. While many different test case selection approaches have been proposed in the past, most techniques share a major drawback: they rely on the availability of source code (or test models) and precise information about modifications to the system [ERS10]. Black-box testing test case selection techniques are sparse and, thus, one focus of this thesis.

One problem that remains with any test case selection approach is that the all selected test cases are to be executed. In other words, even though a subset of test cases has been computed, the number of selected regression test cases might still be too large to be executed in its entirety. To prevent this issue, test case prioritization techniques have been defined.

2.2.2 Test Case Prioritization

Problem Definition. While test case selection aims to find a subset of test cases TC_{retest} to be retested, *test case prioritization* [RUCH01] aims to prioritize a set of applicable test cases $\mathcal{TC}$ for a certain SUT according to their *priority*. Formally, Elbaum et al. [EMR01] define the test case prioritization problem as follows:

Definition 2.2: Test Case Prioritization Problem

Let $TS \subseteq \mathcal{TC}$ be a test set, $\mathcal{P}(TS)$ the set of its permutations and $f : \mathcal{P}(\mathcal{TC}) \to \mathbb{N}$ a priority function.

Problem: Find $TS' \in \mathcal{P}(TS)$ such that $\{\forall TS'' \in \mathcal{P}(TS) \mid TS'' \neq TS' \wedge f(TS') \geq f(TS'')\}$

Test Case Prioritization Techniques. One major advantage of test case prioritization compared to test case selection is that testing can stop at any point in time and, due to the prioritization, we ensure that the most important test cases have been executed up to this point in time. The major goal of test case prioritization is to find faults in the system as early as possible, i.e., executing effective test cases first [RUCH01]. Knowing the optimal permutation for test cases prior to execution is not possible as the information about faults is first available when executing test cases. However, certain heuristics have been developed which aim to maximize a certain goal as early as possible, e.g., by considering code [EMR01, LHH07, YHTS09], historical test data [KP02, FKAP09, ERL11], test models [KKT07] or, rarely, data available in black-box testing [HFM15].

Assessing Prioritization Quality. No matter which technique is used, the desired outcome is to find faults early. To evaluate whether this goal has been achieved, the *Average Percentage of Faults Detected* (APFD) metric [RUCH01] has been defined. APFD returns a value between 0 and 1, where 1 corresponds to the best available result. It computes the detection rate in which faults are covered by test cases in an ordered list. Thus, the earlier a test case at position TF_i reveals the i-th fault, the better is the prioritization. In this thesis, we measure revealing failures instead of faults, as we do not have access to code level information and, thus, only observe fault manifestations in form of failures.

Definition 2.3: Average Percentage of Faults Detected (APFD)

Let $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$ be a ordered test set with n test cases, $\mathcal{F} = \{fail_1, \ldots, fail_m\}$ be a set of m failures and TF_i the i-th position at which the a failure is found with $i \in \{1, \ldots, n\}$.

We define the APFD metric according to Rothermel et al. [RUCH01] as:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \cdot m} + \frac{1}{2n}, n, m > 0$$

Example 2.4: Measuring Test Case Prioritization Quality

Assume the following two permutations of five test cases $\mathcal{TC} = \{tc_1, \ldots, tc_5\}$:

$$TS_1 = (tc_1, tc_2, tc_3, tc_4, tc_5)$$
$$TS_2 = (tc_3, tc_5, tc_1, tc_2, tc_4)$$

Now, further assume that tc_3 and tc_5 are failure revealing test cases. Looking at the two permutations, we see that TS_2 is the superior ordering of test cases as both failures are found by executing the first two test cases. In contrast, TS_1 finds the failures in the third and last test case. Consequently, TS_1 only achieves an APFD value of 0.3 while TS_2 results in an APFD value of 0.8.

2.3 Software Product Lines

Providing Individual Product Variants. Modern software systems are complex and require a high effort due to the increasing functionality. Besides component-based software, another factor which increases the complexity of software systems is the demand for customized software, i.e., systems which can be individually adapted to customer wishes. These variant-rich software systems are described, e.g., as *software product lines* (SPL) [PBvdL05]. An SPL comprises commonalities and variability to allow for the derivation of a finite non-empty set of distinct *product variants* $P_{SPL} = \{p_1, \ldots, p_n\}$. One prominent example for SPLs are modern automobiles. Today, customers are able to configure their car in the ordering process according to their specification and needs. These selectable functionalities increase the number of potential product variants which can be generated.

Challenges in SPLs. A high degree of variability introduces new challenges for software engineering, e.g., dealing with a rising number of product variants in P_{SPL} and redundancy between product variants. Considering the testing of variant-rich software, SPL engineering aims to ensure that each potentially derivable product variant $p \in P_{SPL}$ has been correctly implemented. Due to the rising number of product variants, testing is a complex problem which requires new approaches to reduce testing effort for SPLs compared to traditional testing approaches, which consider each product variant as a new system and require a full testing for each possible product variant [Eng10].

In the following, we explain how SPLs are engineered, how commonalities and variability of SPLs are described and we give insights into delta-oriented modeling.

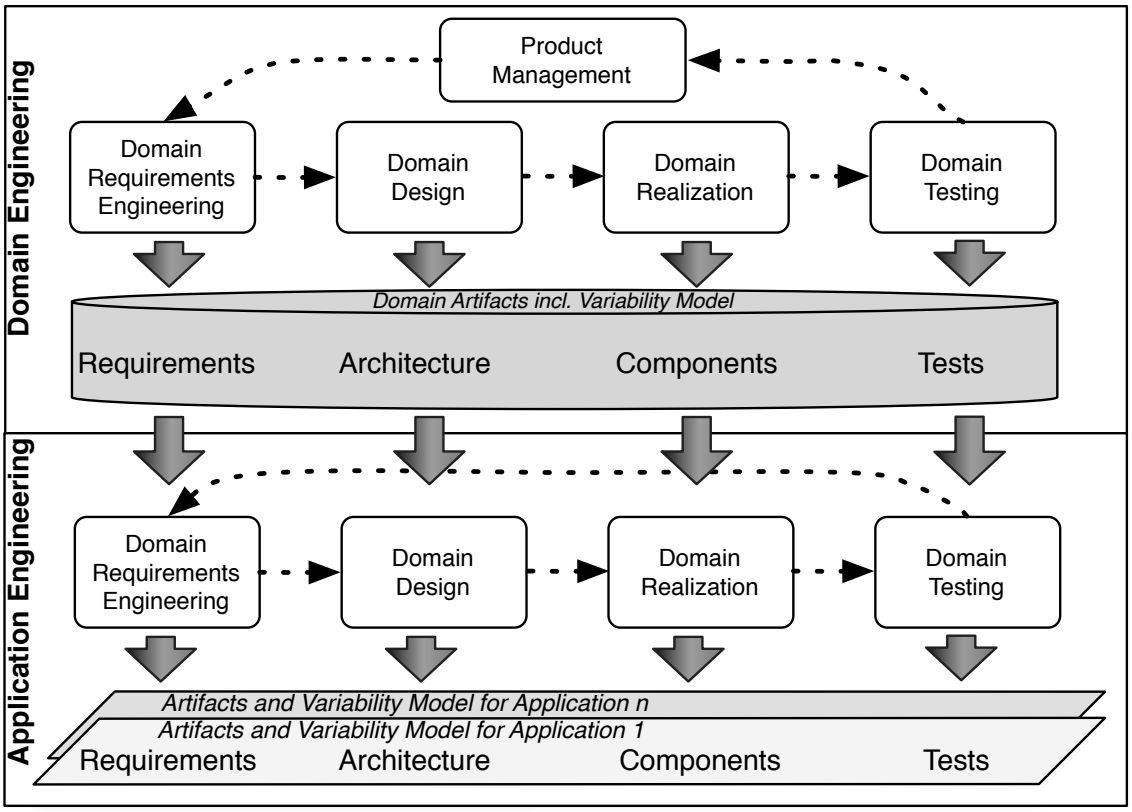

Figure 2.6: The Main SPL Engineering Phases (cf. Pohl et al. [PBvdL05])

2.3.1 Software Product Line Engineering

The SPL engineering process is designed in two main phases: *Domain Engineering* and *Application Engineering* [PBvdL05]. This two phases and their subphases are shown in Figure 2.6 and are briefly described according to Pohl et al. [PBvdL05] in the following.

Domain Engineering. Domain Engineering comprises the necessary steps to create *domain artifacts* which describe the requirements, architecture, components and tests of an SPL as a whole and from which different variants can be extracted from. Its subphases are shown in the upper half of Figure 2.6.

First, the **product management** deals with the economical aspects of SPL engineering. It focuses on market strategies and management tasks related to the SPL engineering process. These decisions and strategies are based on abstract company goals. Product managers decide, based on existing and planned products, which interdependences between products exist and how to define variability. The result

is a *product road map*, including intended product variants and their planned commonalities and variabilities.

Next, common and variable domain requirements are derived in the **domain requirements engineering** subphase based on the product road map. This is a continuous process, which takes the variability of the SPL into account to derive a set of domain requirements. Furthermore, the requirements are distinguished to be either common for all product variants or variant specific. The result is a *variability model*, which comprises different information, e.g., variation points and variants. Similar requirements are grouped together while variable requirements are identified by variation points.

The **domain design** is based on the derived requirements. It comprises the software structure and technical solutions chosen for the architecture based on the variability model derived earlier. The domain design is performed iteratively. Requirements are mapped to technical solutions in a *reference architecture*. The final reference architecture supports the mass customization of product variants. Consequently, the result of this process is a reference architecture and an adapted variability model, which comprises *internal variability*, which is required to fulfill technical requirements.

In the **domain realization** subphase, the previously designed reference architecture is used as a basis to design and implement *reusable software assets*. This includes reusable *components and interfaces* as well as database tables, protocols, etc. This phase also enables configuration mechanisms, i.e., it enables the product variant selection and application realization. The main activity is to build a working system, i.e., design and implementation of the necessary artifacts to receive executable code which can be used in later application realization. Interfaces are crucial to this phase, as they define how components interact with each other and what (variable) functionality a component offers. The resulting components should be robust and configuration independent to achieve reusability.

The fifth subphase in the domain engineering is **domain testing**. This phase aims to validate the output of the other processes involved in the domain engineering, mainly the realization artifacts. Thus, the desired output is a efficient testing process applicable to the domain engineering artifacts using reusable test artifacts. Similar to single-software systems, testing should commence as early as possible and be repeated if necessary. To deal with the variability, the variability model is used to create test artifacts. Single applications are not tested in domain testing, instead, reusable components are the focus of this subphase, i.e., there is no executable system available at this point The goal of this phase is to find faults in domain engineering artifacts.

Application Engineering. The five subphases of the domain engineering lead to a set of *domain artifacts*. They build a platform of the SPL, which is the input for the

application engineering phase. Similar to the domain engineering, the application engineering can be divided into four subphases which correlate to their counterparts in the domain engineering [PBvdL05]. The four subphases are shown in the lower half of Figure 2.6 and are briefly explained in the following according to Pohl et al. [PBvdL05].

To create an application, the **application requirements engineering** subphase is executed first. This subphase aims to reuse domain engineering requirement artifacts and, based on them, define and document requirement artifacts for a certain application of the SPL. Stakeholders are involved in this phase, and their requirements are mapped to variable domain requirements artifacts. In case stakeholder artifacts cannot be satisfied using the domain requirements, additional *application-specific requirements* have to be defined.

After the requirements for an application have been defined, the **application design** subphase is initiated. Here, the *application architecture* is defined, which is a specialization of the reference architecture. Certain application-specific changes are combined with the reusable artifacts of the reference architecture, which saves a lot of time compared to typical single-software architecture design. After an application is finished, architectural changes can be integrated back into the domain artifacts. The resulting application architecture has to fulfill the application's requirements.

The **application realization** takes the architectural design as input and leads to the development of the application. In particular, this sub-phase provides *application-specific components and interfaces*, selected variants of reused components and the application's configuration. The configuration requires the binding of certain variation points in reusable components. As result, an *executable application* is build in this sub-phase. While interfaces are reused from the domain realization, components with internal variability have to be configured.

After an application has been implemented, it is ready for testing. The **application testing** subphase focuses on the quality of the created application and aims to detect faults as early as possible. Application testing is closely related to the domain testing phase by reusing domain test artifacts for multiple applications. This requires *traceability* between requirements and test cases to retrieve the reusable domain test artifacts based on the domain requirements. Similar to single-software testing, different testing phases (cf. Chapter 2.1) and coverage criteria are part of this phase. Optimally, a thorough test of each variant is performed, even though resources are often limited and number of product variants might be excessively large [ER11].

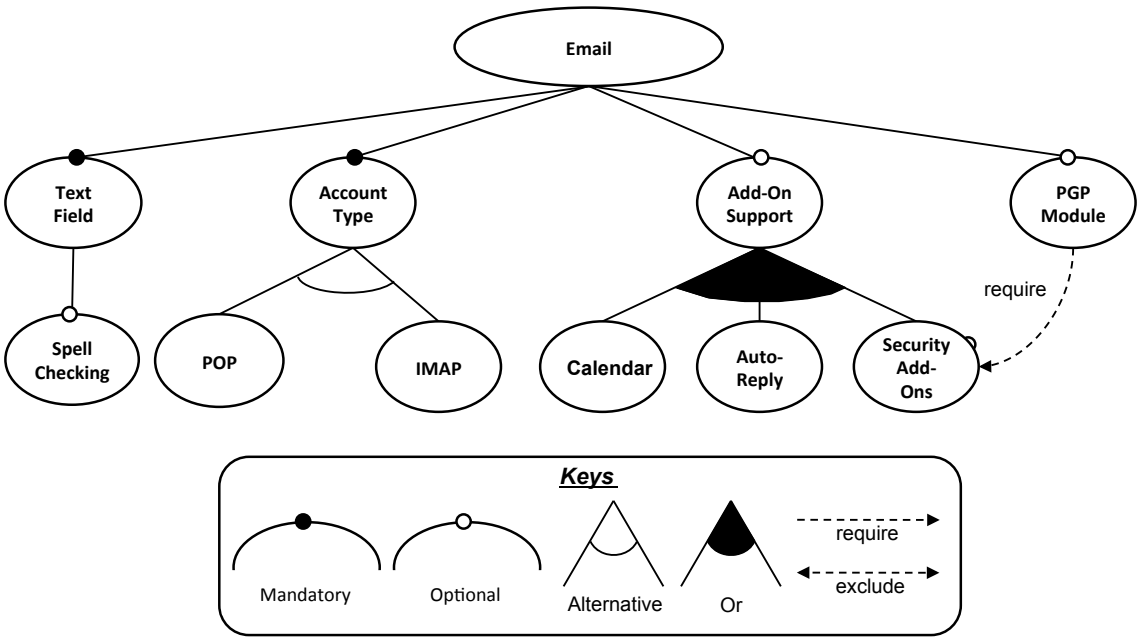

Figure 2.7: A Sample Feature Model Representing an Email-Client SPL

2.3.2 Features and Feature Models

Problem Space Realization. Different techniques exist to describe an SPL in the *problem space* [CE00], i.e., to describe domain-specific concepts derived in the domain engineering [PBvdL05]. One popular approach is the identification and definition of *features*, which, in case of this thesis, describe certain customer-aware functionalities of an SPL. A set of features can be aggregated in different types of variability models. One popular model type are *feature models* [KCH+90] (cf. Figure 2.7), which we also use in this thesis.

Feature Definition. SPLs comprise a finite non-empty set of features $F_{SPL} = \{f_1, \ldots, f_n\}$. Different *types* of features are distinguished to enable the modeling of commonalities and the variability of an SPL [CE00, PBvdL05]. *Mandatory* features are present in every product variant $p \in P_{SPL}$, i.e., they describe the commonalities of an SPL. In contrast, *optional* features describe the variability of an SPL as they can be individually selected by a customer on demand. In addition, variability is introduced by groups of *alternative* features. Here, exactly one of n features in the alternative-group is present in a product variant. Another approach to describe variability in SPLs are *or*-groups, where one to n features can be present in one product variant at the time.

Graphical Feature Models. The types of features are usually distinguished in a graphical fashion. Thus, a feature model fm is depicted as graphical *feature diagram*[SHT06]. They describe SPLs in an easy to grasp fashion, including the involved commonalities and variabilities. Feature diagrams are tree-like graphs, where a feature is a node and their relationships are presented by edges connecting the node. The graphical representation of a node shows if a feature is either mandatory or optional. In addition, two types of arcs represent if features are grouped as *or* or *alternative* features.

Feature Constraints. In addition to features, *cross-tree constraints* describe relationships between features which are not directly connected in the feature model [CE00]. Two simple cross-tree constraints are *require* and *exclude* relationships, which either force one feature to be selected to select another or which forbid to select two particular features at the same time. Additionally, more complex constraints can be formulated using logical formulas over the set of features to define specific aspects of the SPL.

Example 2.5: Email-Client Feature Model

An example for a feature model is shown in Figure 2.7. This feature model represents a basic SPL for a fictional email-client. In this SPL, the text field feature is mandatory. However, the spell checking feature is optional. In contrast, certain add-ons can be selected if Add-On Support is selected. In this case, at least one add-on has to be selected as they are grouped in an *or*-group. Either POP or IMAP have to be selected, but not both at the same time. If the PGP module feature is selected, the Security Add-Ons feature has to be selected as well due to the *requires* cross-tree constraint.

Product Variant Derivation. Using the set of features defined in a feature model, a set of distinct product variants P_{SPL} can be derived. Each product variant corresponds to a particular set of selected features. Furthermore, these feature configurations have to be valid, i.e., they do not violate the constraints of the feature model. For example, for the feature model shown in Figure 2.7, it is invalid to select the features IMAP and POP at the same time as they are restricted by a cross-tree constraint *exclude*. Similarly, child features can only be selected if parent features have been selected for a product variant as well.

2.3.3 Delta-Oriented Modeling

Solution Space Realization. Different techniques [SRC+12] exist to define the *solution space* [CE00] of an SPL, i.e., to describe the definition of reusable software artifacts. Popular approaches to describe the solution space are, e.g., *aspect-oriented programming* [KLM+97], *feature-oriented programming* [Pre97] or *delta-oriented programming* [SBB+10]. These techniques allow to implement SPLs and, thus, are able to describe the variability of these types of systems. We focus on *delta modeling* [CHS10] in this thesis, which is a language and model type independent transformational approach to model SPL artifacts.

Delta Definition. Due to its transformational nature, delta-oriented modeling requires that a certain variant of the SPL is selected to be the *core variant*. The core should be valid variant of the SPL holds. Based on the core variant, the set of remaining product variants in the SPL can be derived by defining a set of *deltas*. Each delta comprises a set of *delta operations*, which define transformations applicable to the core variant. A valid delta operation either *adds*, *removes* or *modifies* elements of the core variant to generate other product variants [CHS10]. The mapping between a configuration and required deltas is provided by an *application condition* assigned to each delta. In case of this thesis, we assume that an application condition is a boolean constraint over the set of features of an SPL.

Delta Application. To generate the model for a certain product variant, the application conditions of all deltas are evaluated. If an application condition is evaluated to `true`, the corresponding delta operations are applied to the core variant. Of course, more than one delta might have to be applied to generate a certain product variant. Hence, it is also important to validate the *ordering* of deltas, i.e., one delta potentially has to be executed before another delta to receive a valid product configuration. For example, an element can only be modified or removed if it is already present in the current model. While the ordering of deltas is important for the construction of product variants, we do not focus on this issue later on and assume it is resolved by the designer using an *after*-condition for each delta, defining if other deltas have to applied before to ensure that the resulting product variant is valid. Delta modeling has been applied to different types of artifacts, e.g., *state machines* [LSKL12], architectures [LLL+14], requirements [DSLL13] or Java code [KHS+14]. In context of this thesis, delta-oriented state machines and architectures are relevant as we use them to evaluate our black-box testing techniques for software variants.

3 Foundations

This thesis presents techniques to improve black-box testing of software variants and software versions. We explain used testing artifacts in this chapter. Furthermore, we introduce the *Body Comfort System* (BCS) [LLLS12] case study which we use to evaluate the quality of our developed black-box testing techniques.

3.1 Test Artifacts

To apply our black-box testing techniques for software variants and versions, we require certain artifacts to be available for testing. They are the foundation for the black-box regression testing techniques for variants and versions. We assume data is stored in a test management system (or similar), e.g., HP Quality Center[1], and that data is machine-processable. Figure 3.1 shows an overview of the type of data for which our black-box testing techniques are (partially) applicable. We distinguish between artifacts for software variants and software versions, which we explain in detail in the following.

3.1.1 Artifacts for Testing of Software Variants

This thesis presents techniques to test software variants in variant-rich systems, such as software product lines (SPL). We assume that the system under test is a black-box, i.e., no source code is available. Thus, in the context of this thesis, we perform a model-based testing approach for testing of software variants. Test models are used as a specification to derive test cases, which are executed for the system.

In the following, we first show the foundations of our SPL descriptions. We give an overview of the SPL-specific artifacts in Table 3.1. This comprises feature models and a general definition of abstract test models. In addition, we present specific delta-oriented test used for integration testing. Based on this, we define how integration test cases are described, which we use in the course of this thesis as evaluation data for our black-box testing techniques for software variants.

SPL Description. We first require a description of the SPL under test. As described in Chapter 2.3.2, feature models [KCH+90] are a common technique to

[1] HP Quality Center is part of the HP Application Lifecycle Management (ALM). Website: `http://www8.hp.com/us/en/software-solutions/application-lifecycle-management.html`

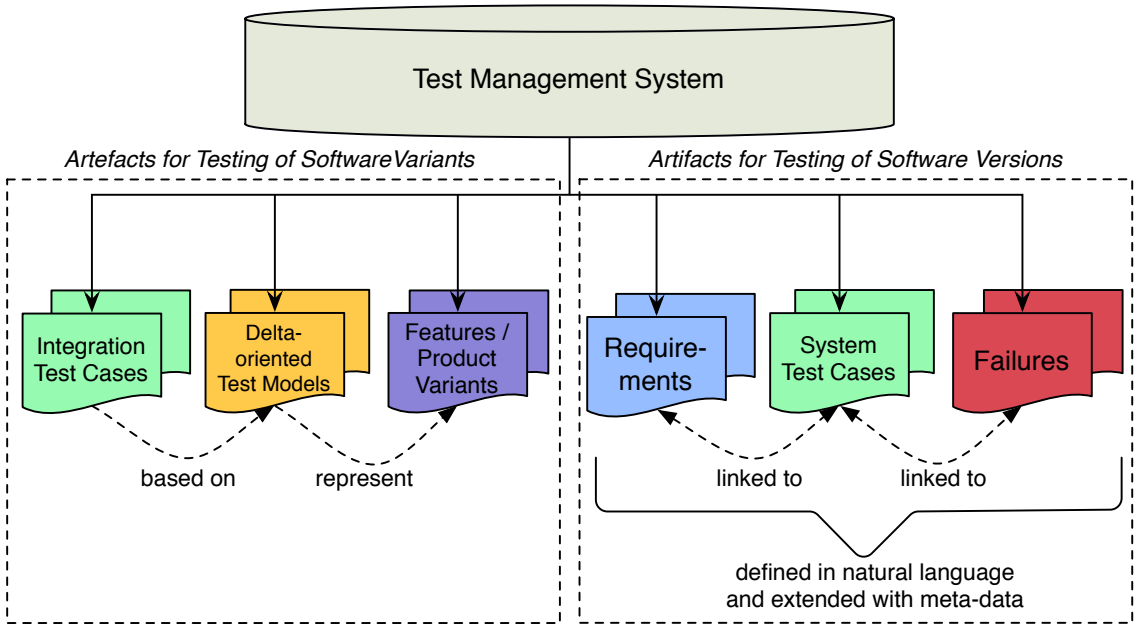

Figure 3.1: Test Artifacts utilized for Black-Box Testing of Variants and Versions

describe a set of product variants. In this thesis, a feature model fm defines a finite non-empty set of features $F_{SPL} = \{f_1, \ldots, f_n\}$, which describe the space of valid product variants. We define a finite non-empty set of valid product variants as $P_{SPL} = \{p_1, \ldots, p_m\}$, i.e., a valid product variant represents a feature configuration which satisfies the feature model constraints.

Abstract Test Models. Our SPL testing techniques are model-based [UL07]. They analyze test models to guide SPL testing. Each test model represents one product variant. The presented techniques are very flexible and do not require specific test models, i.e., we use abstract test models as foundation for our black-box testing approach for software variants, retracting from specific test model types to keep our techniques flexible. However, applied test models have to comply with certain characteristics. In this thesis, test models syntactically represent a directed, connected graph, comprising a finite set of vertices and edges.

Definition 3.1: Abstract Test Model

We define an *abstract test model* as a connected graph, consisting of a finite non-empty set of *vertices* $V = \{v_1, \ldots, v_n\}$ and a finite non-empty set of *edges* $E \subseteq V \times V$ as relation between two vertices. Each edge $e = (v, v')$ connects a vertex v to vertex v' in a directed fashion.

Table 3.1: Testing Artifacts for Software Variants

Name	*Formal Definition / Signature*
SPL Related	
Feature model	fm
Set of features	$F_{SPL} = \{f_1, \ldots, f_n\}$
Set of product variants	$P_{SPL} = \{p_1, \ldots, p_m\}$
Test Model Related	
Set of vertices	$V = \{v_1, \ldots, v_n\}$
Set of edges	$E \subseteq V \times V$
Vertex neighborhood	$NC : V \times P_{SPL} \to \mathcal{P}(V)$
Vertex Interface function	$Int : V \times P_{SPL} \to \mathcal{P}(E)$
Test Model Element	$\Omega = \{\omega_1, \ldots, \omega_k\} = V \cup E$
Delta Related	
Set of deltas	$\Delta_{SPL} = \{\delta_1, \ldots, \delta_m\}$
Core variant	$p_{core} \in P_{SPL}$
Set of delta operations	$\mathcal{OP} = \{op_1, \ldots, op_n\}$
Set of delta operations for product variant	$operations : \Omega \times P_{SPL} \to \mathcal{P}(\mathcal{OP})$
Delta application condition	$\varphi \in \mathbb{B}(FS)$
Regression Delta	$\delta_{p_i,p_j}, p_i, p_j \in P_{SPL}$
Number of Multi Product Deltas for ω	$MPD_\omega \in \mathbb{N}$

We do not specify the semantics of vertices in abstract test models to allow our testing techniques to be applied in different testing phases and on different granularity-levels. For example, in unit testing, behavioral models are prominent to specify the correct functionality of single components [UL07]. Here, the set of abstract vertices V represents a set of *states* S the system can be in and the edge-relation represents a set of *transitions* T between states. In integration testing, architectural models are used to specify the correct communication of components. In this testing phase, the set of test model vertices represents a set of *components* and the relations E between components describe a connector relation $CON = C \times \Pi \times C$. A connector transfers a signal $\pi \in \Pi$ between components.

Vertices are always connected to at least one other vertex via at least one *edge* $e \in E$. Edges are directed, i.e., an edge $e = (v, v')$ connects vertex v to vertex v'. For this thesis, connected vertices are of interest as the underlying system parts might influence each other. We compute the *neighborhood* of a vertex based on the connected incoming and outgoing edges.

Definition 3.2: Vertex Neighborhood Function

Let $G = (V, E)$ be a graph consisting of a set of vertices V and edges E corresponding to a product variant $p \in P_{SPL}$, V_p the set of vertices in product variant p and $e = (v, v') \in E$ an edge from vertex v to vertex v'.

The set of vertices, which are connected to a vertex $v \in V_p$ for product variant $p \in P_{SPL}$ is computed using the function $NC : V \times P_{SPL} \to \mathcal{P}(V)$ as follows:

$$NC(v, p) = \{v' \in V_p \mid \exists\, e \in E : e = (v, v') \vee e = (v', v)\}$$

We define a *vertex interface function* $Int : V \times P_{SPL} \to \mathcal{P}(E)$, which returns the set of edges connected to a vertex. This function is important in the context of integration testing, where vertices resemble components and edges represent connectors, as integration testing focuses on the communication of different components via their interfaces. In other words, the correct use of a component interface is tested.

Definition 3.3: Vertex Interface Function

Let $G = (V, E)$ be a graph with a set of vertices V and a set of edges E representing a product variant $p \in P_{SPL}$, V_p the set of vertices for a particular product variant and $e = (v, v') \in E$ be an edge from vertex v to v'.

We define the *vertex interface function* $Int : V \times P_{SPL} \to \mathcal{P}(E)$ to compute the set of edges, which are part of the interface of vertex $v \in V_p$ in a product variant $p \in P_{SPL}$ as:

$$Int(v, p) = \{e \in E_p \mid \exists\, v \in V_p : e = (v, v') \vee e = (v', v)\}$$

In the context of this thesis, we refer to the set of vertices and edges of a test model as the set of *test model elements* $\Omega = V \cup E$, ranging over $\omega, \omega_1, \omega_2, \ldots$

Deltas. As explained in Chapter 2.3.3, deltas describe transformations to generate product variants of an SPL [CHS10]. In context of this thesis, they also support the identification of important differences between individual product variants. In case of our model-based testing technique, the set of deltas $\Delta_{SPL} = \{\delta_1, \ldots, \delta_m\}$ transform test model elements $\omega \in \Omega$. Deltas comprises a finite non-empty set of delta operations $\mathcal{OP} = \{add\ \omega, remove\ \omega, modify\ \omega \mid \omega \in \Omega\}$. Additional, model type specific transformations are also allowed, e.g., deltas for architectural models are able to transform signals.

Definition 3.4: Delta Operations for Test Model Elements

Let $\Omega = (V \cup E)$ be the set of test model elements in a graph $G = (V, E)$, which represents a product variant $p \in P_{SPL}$, $\Delta_p \subseteq \Delta_{SPL}$ the set of deltas applied to generate product variant $p \in P_{SPL}$ and $\mathcal{OP} = \{op_1, \ldots, op_n\}$ the set of delta operations.

We define the *delta operations* function $operations : \Omega \times P_{SPL} \rightarrow \mathcal{P}(\mathcal{OP})$ which retrieves a set of delta operations for a given test model element $\omega \in \Omega$ in a product variant $p \in P_{SPL}$ as follows:

$$operations(\omega, p) = \{op' \in \mathcal{OP} \mid op' \in \{add\ \omega, remove\ \omega, modify\ \omega\} \wedge \exists\, \delta \in \Delta_p : op' \in \delta\}$$

Each delta has an *application condition* φ, which defines for which product variant a delta is to be applied. In this thesis, we define an application condition φ for a delta as a boolean formula over the set of features $FS \subseteq F_{SPL}$, denoted as $\mathbb{B}(FS)$, i.e., it holds that $\varphi \in \mathbb{B}(FS)$ for valid application conditions. If the application condition of a delta is fulfilled for a certain product variant, the corresponding delta operations are applied to the test model. We define a function $conditions : \Delta_{SPL} \rightarrow \mathcal{P}(F_{SPL})$, which returns the set of features contained in the application condition of a set of deltas, which are also part of the product configuration for the current product variant p. In other words, the function does not return negated features, as these are not selected in the product variant and, thus, do not map to any test model elements for this particular product variant. Instead, we are interested which features of the product variant influence which parts of the system, e.g., by adding or removing them.

Delta-Oriented Test Models for Integration Testing. In the context of this thesis, we do not focus on delta modeling or delta generation, i.e., we do not provide guidance on how to design a delta-oriented SPL. Instead, we assume that deltas are either manually designed by an expert or extracted from existing product variants, e.g., by using delta generation techniques [PKK+15, WRSS17].

Our testing techniques for SPLs require delta-oriented test models. We do not restrict our testing technique to certain types of delta-oriented test model. In context of this thesis, we show the applicability of our testing approaches for software variants in integration testing. Thus, we use *delta-oriented architectural test models* as an instance for our abstract test model notation to show the applicability of our SPL testing techniques. Here, the set of vertices is a set of components $C = \{c_1, \ldots, c_n\}$ and the edges describe a set of connectors $CON = \{con_1, \ldots, con_m\}$ in the architecture of a product variant (cf. Chapter 2.1.3). Thus, the interface of a component

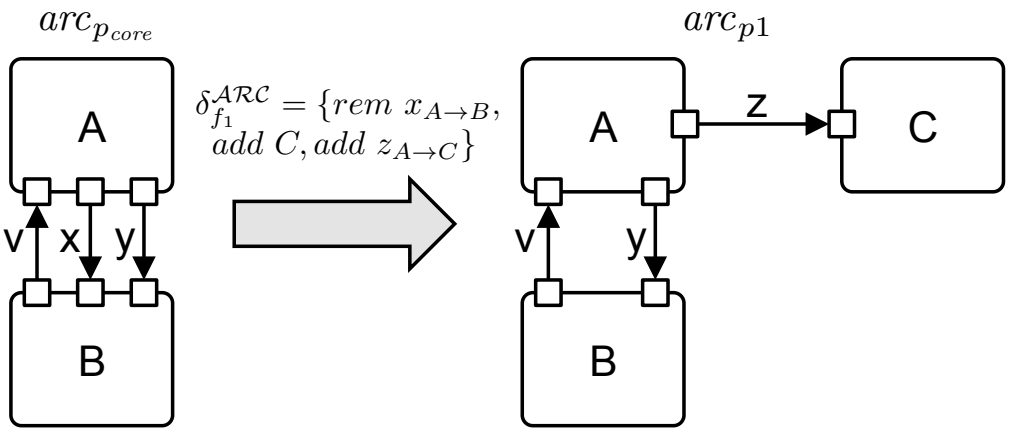

Figure 3.2: Sample Delta-Oriented Modification of Architecture Model

is the set of incoming and outgoing connectors. For architectural models, we define a set of deltas $\Delta_{SPL}^{\mathcal{ARC}} = \{\delta_1^{\mathcal{ARC}}, \ldots, \delta_m^{\mathcal{ARC}}\}$. Each delta consists of a set of *delta operations* $OP \subseteq \mathcal{OP}$, where $op \in \{add\ \omega, remove\ \omega, modify\ \omega\}$ for test model elements $\omega \in (C \cup CON \cup \Pi)$. An architectural delta operation either *adds*, *removes* or *modifies* components, connectors or signals of an architecture test model. We require the results of a delta application to still be valid test models, otherwise the deltas have not been correctly designed or applied.

Example 3.1: Delta Modification of Architecture Models

Figure 3.2 shows the derivation of the architectural model arc_{p_1} for product variant p_1 based on the architectural model of variant p_{core}. Assume that the shown delta $\delta_{f_1}^{ARC}$ is applied for a certain feature f_1, which is not further specified. The delta contains three delta operations. The first operation removes the connector between A and B, which transfers signal x. The other two delta operations first add a new component C and then a connector between A and C, which transfers the new signal z. The application of these three delta operations leads to the product variant p_1 shown on the right-hand side of Figure 3.2.

Similar to architecture models, we apply our SPL testing technique to behavioral test models in the context of SPL testing. These types of test models describe the internal behavior of components in the system in a state-based fashion [UL07]. In particular, we utilize *delta-oriented state machines* to describe the behavior of components in different product variants of the SPL. Here, we instantiate our abstract test model notation as follows: vertices represent the finite set of states $S = \{s_1, \ldots, s_n\}$ and the relations between states are defined as finite set of *transitions* $T = S \times L \times S$,

where L is the set of transition labels (cf. Chapter 2.1.3), containing an incoming event $\Sigma_I \in \Sigma$ and a set of outgoing events $\Sigma_O \subseteq \Sigma$. Transitions are triggered if a certain event $\sigma \in \Sigma$ is fulfilled (cf. Chapter 2.1.3). We define a set of state machine deltas $\Delta_{SPL}^{\mathcal{SM}} = \{\delta_1^{\mathcal{SM}}, \ldots, \delta_m^{\mathcal{SM}}\}$. A delta contains a set of delta operations $\mathcal{OP} = \{op_1, \ldots, op_k\}$. Each delta operations either adds, removes or modifies states, transitions or events., i.e., $op \in \{add\ \omega, remove\ \omega, modify\ \omega\}$, where $\omega \in (S \cup T \cup \Sigma)$ holds.

As for architecture models, we require the state machine models to be valid. For example, assume the architecture model on the right-hand side of Figure 3.2. While this particular architecture model is valid, it would become invalid if a delta would remove connector z, which splits the graph into two unconnected subgraphs.

Regression Deltas. Besides the application of deltas describing the SPL, we compute *regression deltas* between two product variants [LSKL12]. A regression delta $\delta_{p,p'}$ describes the transformations required to generate a variant p' based on another variant p. The computation of these deltas is based on the predefined delta set Δ_{SPL}. We compute the symmetrical difference for deltas applied to two product variants p and p', where $p \neq p'$. To this end, we define the *inverse* of a delta operation as $\delta^{-1} \subseteq \mathcal{OP}$, i.e., $\{add\ \omega\}^{-1} = \{rem\ \omega\}$ and $\{rem\ \omega\}^{-1} = \{add\ \omega\}$. We compose the inverse delta of one product variant with the delta of another, ignoring operations present in both product variants. This leads to the regression delta computation [LSKL12] in Definition 3.5.

Definition 3.5: Regression Delta Computation

Let $\delta_p \in \Delta_{SPL}$ be the delta comprising the delta operations applied for product variant $p \in P_{SPL}$ and $\delta^{-1} \subseteq \mathcal{OP}$ the *inverse* of a set of delta operations.

According to Lochau et al. [LSKL12], we define the regression delta between two product variants p and p' based on their symmetrical difference as follows:

$$\delta_{p,p'} = (\delta_p \setminus \delta_{p'})^{-1} \cup (\delta_{p'} \setminus \delta_p)$$

Regression deltas are useful for incremental testing techniques [LSKL12]. In this context, we step from one product variant to the next and reveal what has changed compared to previously tested variants $P_{tested} \subseteq P_{SPL}$. This can be exploited for testing delta-oriented SPLs by focusing on the differences between product variants under test [LLL+14]. Our SPL testing techniques use regression deltas to identify never before tested changes in product variants.

Multi Product Deltas. In certain circumstances, occurring changes between product variants are not detectable when analyzing the deltas of a product variant.

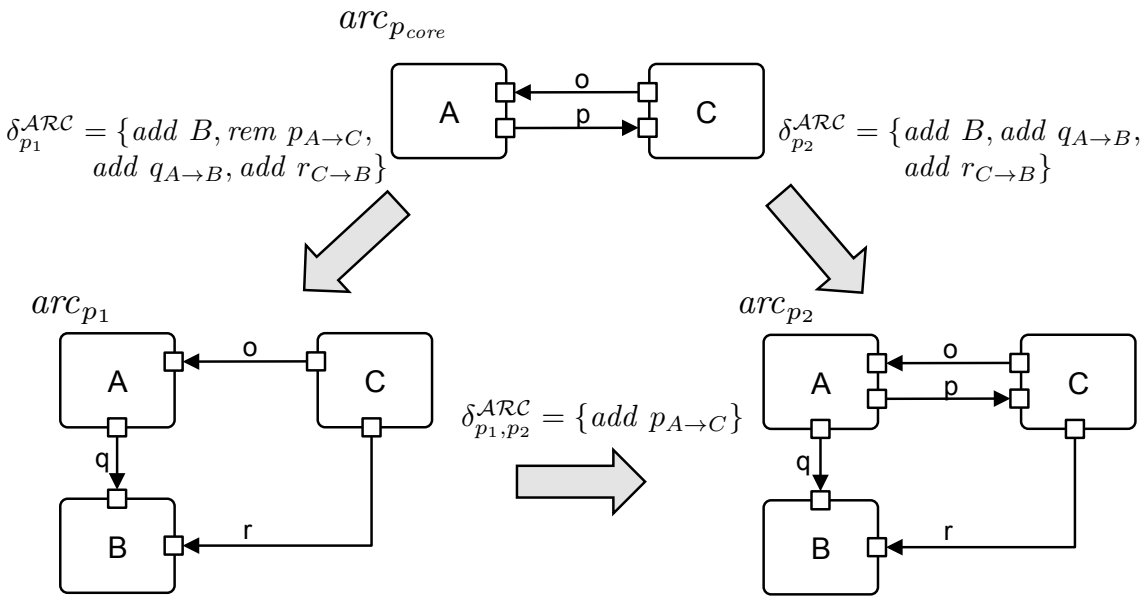

Figure 3.3: Sample Multi Product Delta Scenario (cf. [LLL+15])

We refer to these types of changes as *multi product deltas* (MPD) [LLL+15] as they are only detectable when comparing the deltas of the current product variant with several other product variants. In particular, MPDs might occur when all deltas related to a certain test model element have been applied before to previously tested product variants $P_{tested} \subseteq P_{SPL}$, where the index of a product variant represents its position in the testing process, i.e., $p_1 \in P_{tested}$ has been tested first. In other words, the deltas for a current *product variant under test* (PUT) all have been applied to previously tested product variants, but never in the particular combination present in the current PUT. This might lead to never before tested system configurations and, thus, changes between product variants. When analyzing test models, an MPD occurs if the interface (i.e., incoming and outgoing edges) of a vertex has never occurred in previous product variants for the particular vertex. This situation occurs, if all deltas related to this vertex have been applied before, but never in the particular combination present for the current PUT. We define $MPD_\omega \in \mathbb{N}$ to be the number of MPDs for a certain test model element in the current PUT. We argue that MPDs should be tested as well to ensure the correct behavior of a product variant.

Example 3.2: Multi Product Deltas in Architecture Test Models

Figure 3.3 shows architecture models for three product variants p_{core}, p_1 and p_2 as well as the deltas to derive p_1 and p_2 and their regression delta $\delta^{\mathcal{ARC}}_{p_1,p_2}$. Assume that an incremental testing process is performed and the current PUT is p_2, i.e., $P_{tested} = \{p_{core}, p_1\}$. We compute the novel regression delta operations for p_2 based on Definition 4.1, resulting in $\delta^{new}_{p_2} = \emptyset$ as each delta operation of p_2 has either occurred in δ_{p_1} or has been covered in the core. However, when we analyze the component interfaces in p_2, we notice that component C is present in a never before tested interface configuration, as all three connectors o, p and r are present for the first time in unison. This indicates that a *multi product delta* exists, i.e., a change that is only observable when looking at all previously tested variants and the already tested interface configurations. Thus, $MPD_C = 1$, as the smallest difference to all other variants of C is one connector. Hence, we argue that component C is of interest for testing.

Integration Test Cases. Our SPL testing techniques aim to prioritize test cases for product variants under test. In general, a set of test cases $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$ has to be provided for testing. While we do not restrict our SPL testing technique to certain types of test cases, they have to be defined according to the test models used as specification. Thus, test cases cover a set of vertices $V_{tc} \subseteq V$ and a multi-set of edges between these vertices, returned by the function $cov_e : V \times V \to \mathcal{P}(E)$. As we use integration testing to evaluate our testing techniques for product variants, we explain integration test cases in the following.

For integration testing, we assume test cases to be defined as *message sequence charts* (MSC) [HT03]. MSCs are based on architecture models and describe interaction scenarios between different components of the system. Similar to the V-model, we assume that unit testing has been performed before integration testing commences, i.e., we focus on the communication between at least two components. An MSC describes how two or more covered components communicate via exchanged signals, i.e., they describe an ordering of signal exchanges. In general, MSCs describe certain communication scenarios in the system. These scenarios correlate to the defined architecture and component interfaces. The internal behavior of involved components is not visible on this level.

We assume that a finite non-empty set of integration test cases $\mathcal{TC}_{int} = \{tc_1, \ldots, tc_n\}$ is defined for an SPL. An MSC is applicable to a certain set of product variants, which contains the tested behavior. If the MSC has been executed for a previous product variant and is applicable for the current product variant p, we assume that the MSC is in the set of reusable test cases $TC_{reuse_p} \subseteq \mathcal{TC}$ (cf.Chapter 2.2.1).

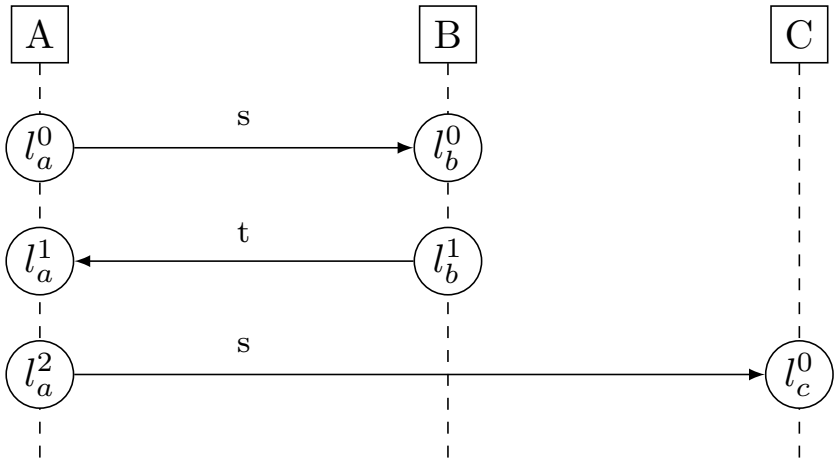

Figure 3.4: Sample Message Sequence Chart

Each test case $tc \in \mathcal{TC}_{int}$ is designed as MSC and represents an interaction scenario between a finite set of components $C_{tc} \subseteq C$ based on a set of covered connectors $CON_{tc} \subseteq CON$, which exchange signals $\pi \in \Pi$ between two components each. While an MSC describes signal exchanges in a certain order to describe an interaction scenario, we do not analyze temporal orderings of covered elements in this thesis as we focus on structural analyses based on architecture definitions.

To analyze the content of an MSC, we define a function $s : C \times C \rightarrow \mathcal{P}(\Pi)$ which returns the multi-set of signals exchanged between two components via their connectors. We define the result as multi-set, as multiple occurrences of the same signal can be specified in a test case. In this thesis, we assume that two test cases are different if the number of occurrences of their shared signals is not identical. For example, if a test case exchanges a signal a two times between two components c_1 and c_2 the result of $s(c_1, c_2)$ is $\{a, a\}$. Another test case which covers signal a only once might describe a different scenario and is considered to be non-identical, e.g., because a might have to occur twice to enable a certain functionality in the system.

Example 3.3: Message Sequence Chart

Figure 3.4 illustrates a sample MSC. This particular test case tc tests the interaction between three components, i.e., $C_{tc} = \{A, B, C\}$. To analyze the exchanged signals, we call function $s(tc)$ which returns the multi-set $\{s, s, t\}$ as signal s is exchanged twice. In particular, the test case is initiated with sending the signal s from A to B. B answers with signal t, which leads A to send signal s to C. Only if we are able to observe the described interaction after sending the initial signal s the test case is considered to be successful.

Table 3.2: Testing Artifacts for Software Versions

Name	*Formal Definition / Signature*
Requirements Related	
Set of system requirements	$\mathcal{REQ} = \{req_1, \ldots, req_n\}$
Test Case Related	
Set of system test cases	$\mathcal{TC}_{sys} = \{tc_1, \ldots, tc_n\}$
Set of covered requirements	$Req_{tc} \subseteq \mathcal{REQ}$
Set of failure revealing test cases	$\mathcal{TC}_F \subseteq \mathcal{TC}_{sys}$
Last test execution function	$lastExec : \mathcal{TC}_{sys} \rightarrow [1, 6]$
Test case execution cost function	$cost : \mathcal{TC}_{sys} \rightarrow \mathbb{N}$
Failure Related	
Set of failures	$\mathcal{F} = \{fail_1, \ldots, fail_m\}$
Failure importance function	$prio : \mathcal{F} \rightarrow \mathbb{N}$
Set of failures detected by a test case	$fail_{tc} \in \mathcal{F}, tc \in \mathcal{TC}_{sys}$

3.1.2 Artifacts for Black-Box Software Versions

In addition to testing of software variants, this thesis also contributes in regression testing of software versions. In particular, we focus on regression testing of versions in system testing. System testing is concerned with the testing of the overall system behavior based on the system specification [Som10] (cf. Chapter 2.1.1). In the following, we assume that the system under test is a black-box and no source code knowledge is available, which is typical for component-based systems [Wey98]. We assume the following data to be available for system testing: *Requirements*, *system test cases* and reported *failures*. An overview of the artifacts and their formal definitions and related functions is given in Table 3.2. This thesis considers failures in system testing instead of faults, as black-box testing does not detect any code-related faults, but observable failures. Furthermore, the system test artifacts are assumed to provide traceability, i.e., requirements are linked to test cases and test cases are linked to the failures they detected. We explain the differences and commonalities of the system testing artifacts and their formal representation used in the context of this thesis in the following.

System Requirements. The system-level specification of the system under test is defined as a finite non-empty set of requirements $\mathcal{REQ} = \{req_1, \ldots, req_n\}$. In literature, usually *functional* or *non-functional* requirements are distinguished [Som10]. In this thesis, we focus on functional requirements, which we assume to be tested manually. A requirement always has a creation date and a description. The description is defined in natural language, i.e., it is written by a human expert. Additional

Table 3.3: Last Test Execution Scale

Value	**1**	**2**	**3**	**4**	**5**	**6**
Since (Days)	≤ 7	≤ 14	≤ 30	≤ 90	> 90	never

information can be linked to requirements, e.g., risk values assigned in a special risk assessment phase during the early development phases [FR14]. Furthermore, we assume requirements coverage, i.e., each requirements is covered by at least one system test case.

System Test Cases. Test cases are the basis of any testing process. In this thesis, we assume a finite non-empty set of system test cases $\mathcal{TC}_{sys} = \{tc_1, \ldots, tc_n\}$ is defined for a particular system under test. Each test case tc is an executable procedure defined for a certain subset of requirements $Req_{tc} \subseteq \mathcal{REQ}$. If a requirement is not linked to a test case, there is no knowledge if the requirements has been correctly implemented. Hence, requirements coverage is a baseline for system-level testing to gain some level of trust into the system under test without code information [WRHM06]. Each system test case $tc \in \mathcal{TC}_{sys}$ is defined by a unique name, has a status (passed or failed), a creation date and a description, which contains the expected result. We assume test cases are defined in natural language, as each test case is assumed to be executed manually by a tester. Each system test case has a *precondition* to be fulfilled to successfully execute the test case. In addition, a set of *test steps* are described, which resemble the actions to be performed by the tester. An *expected result* describes the desired output of test case, i.e., its correct result. If the observed result of a test case differs from its expected result, we consider the test case to be failed. The average execution times are reported by the test management system. We define a function $cost : \mathcal{TC}_{sys} \to \mathbb{N}$, which returns the cost of a test case defined as time in seconds.

In addition to the execution time, the last execution of a test case is tracked in days. To reduce the overhead in dealing with continuous time values in computations, we introduce a predefined discrete 6-point scale based on the time since the last execution. The scale is shown in Table 3.3. Test cases, which have never been executed at all get the highest possible value (6), as they should be tested to achieve a high test coverage. Formally, we define the function $lastExec : \mathcal{TC}_{sys} \to [1, 6] = \{x \in \mathbb{N} \mid 1 \leq x \leq 6\}$ to return the last test execution information.

A test case's *test case description* (TCD) contains different test steps, which are to be executed to achieve a defined expected result. Each test case has an expected result, also defined in natural language. After executing a test case, the tester has to decide whether or not the system under test fulfills its specification, i.e., if the observed behavior conforms to the expected result.

Example 3.4: System Test Case

We assume that a system test case has a precondition, action and expected result. Assume the following definition for a test case $tc \in \mathcal{TC}_{sys}$ to test a fictive webshop:

Precondition: Website is opened in browser, user is not logged in.
Test Steps: Enter *testuser* credentials in the login window and click the *submit* button.
Expected Result: User is logged in and new page *user profile* has opened in browser.

Failures. While testing the system, a set of failures $\mathcal{F} = \{fail_1, \ldots, fail_m\}$ is discovered. If a test case has failed, the produced failure $fail_{tc} \in \mathcal{F}$ is documented and linked to the respective test case. The set of failure revealing test cases is denoted as $\mathcal{TC}_F \subseteq \mathcal{TC}$. This thesis does not focus on fixing of bugs. Instead, we use failures as information for regression testing. For example, test cases which have produced a failure in the past might have a higher chance to do so again. Additionally, reported failures have to be retested after they allegedly have been fixed.

Failures can differ in their importance. Different failure priority definitions can be used, e.g., by introducing a differentiation between A, B and C failure priorities. Here, the priority value A describes critical failures, which have to be fixed before release, failures with B priority will be noticed by some customers and might lead to additional costs and a C priority describes failures of cosmetic nature [Ema05]. We define a function $prio : \mathcal{F} \to \mathbb{N}$, which returns the failure priority of a failure $fail \in \mathcal{F}$ as a discrete numeric value, i.e., we require failure probabilities to be defined on a fixed numeric scale, e.g., from 1 to 5. Different failure priorities indicate different levels of importance regarding the need of correction. In addition to the importance of the failure, we also assume to know the date when the failure was found and fixed.

3.2 Body Comfort System Case Study

Introduction to BCS. The Body Comfort System (BCS) case study describes the body comfort system of a fictional automobile. In its original form, the case study describes a single software system [MLD+09]. BCS comprises different connected *electronic control units* (ECU), such as a *Human Machine Interface* (HMI) which contains several status LEDs and a display to show the current state of the car. A door ECU contains a *power window* including a *finger protection*, which avoids the closing of the window while an object is detected inside. An *heatable exterior*

mirror can be controlled via the door ECU and is enabled if the outside temperature is below a certain threshold value. A *central locking system* protects the car from intruders and be controlled via HMI or a pair of *remote control keys*. An *alarm system* can be activated to signal if a break-in attempt has been noticed while the car was locked. This is complemented by an *interior alarm system*, which can be activated by the user to alert movement within the car.

The BCS SPL. The single-software functionalities have been transformed by Oster et al. [OZLG11] into an SPL, representing a wide variety of possible product variants. The variability and commonalities are captured as a feature model shown in Figure 3.5 (cf. Chapter 2.3.2). The different subsystems have been assigned to a variety of mandatory, optional and abstract features. In total, 27 features have been defined which represent the different product variants of the BCS SPL.

BCS Features. As the feature model in Figure 3.5 shows, certain features of BCS are optional, e.g., the heatable feature of the exterior mirror. Other features, such as the HMI, are present in every variant, i.e., they are mandatory. In addition, at least one status LED has to be selected if the parent feature is selected, i.e., they are assembled in an *or*-group. The manual power window and automatic power window features are alternatives, i.e., exactly one of both has to be selected in the current product variant. There are only few cross-tree constraints in BCS, e.g., the Control Alarm System feature requires the Alarm System feature to be available. A total of 11,616 valid product variants can be configured based on the feature model shown in Figure 3.5.

Product Variant Sampling. SPLs lead to problematic scenarios which are described as *feature interactions* [CKMRM02]. Feature interactions can be of positive or negative nature and occur, if two or more features are combined together. SPL testing aims to ensure that feature interactions do not lead to unexpected or critical behavior. Thus, a lot of work has been put into the detection of feature interactions in the past. One goal is to reduce the number of product variants, shifting the focus to those which are most likely to provoke potential feature interactions [OMR10, ASW+11, JHF12]. Due to the large space of possible product variants in BCS, we use a subset of product variants derived by Oster et al. [OZLG11] using their pairwise feature selection approach *MoSo-PoLiTe*. They ensure that pairwise feature combinations are covered [CKMRM02]. For BCS, Oster et al. [OZLG11] derived 17 product variants for pairwise testing [LLLS12]. We use these product variants in our evaluation to avoid the testing of all product variants.

Delta-Oriented Adaption. In this thesis, we introduce novel delta-oriented testing approaches for SPLs. Lity et al. [LLSG12] transformed BCS into a delta-oriented SPL based on the SPL definition. The authors added a product variant *P0* which serves as the core variant of the SPL. It contains the alternative feature *manual power window* and only mandatory features otherwise. Lity et al. [LLLS12]

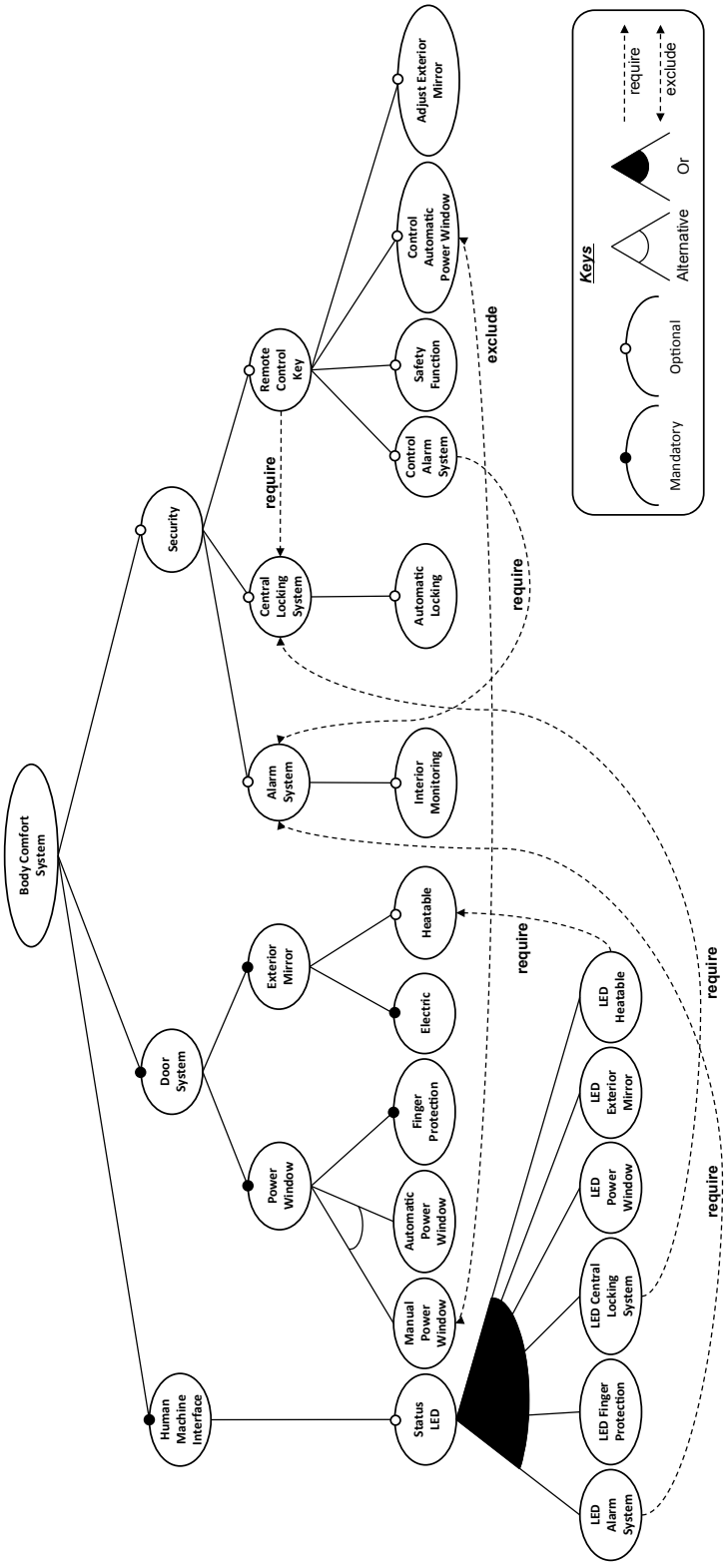

Figure 3.5: The Body Comfort System Feature Model (cf. [LLLS12])

also define deltas to generate each possible variant of the SPL based on the core. These transformations are applied to behavioral and structural test models, which describe different aspects of the product variants.

3.2.1 Delta-Oriented Test Models

The deltas for BCS are defined for test models. First, behavioral deltas have been defined [LSKL12], i.e., they describe the product variants as *state machines* (or state charts) [Har87] (cf. Chapter 2.1.3). In later work, Lity et al. [LLLS12] added architectural descriptions of the SPL to describe the system architectures of product variants.

Delta-Oriented State Machines. The state machines provided for BCS are derived from a 150% model [LLLS12]. They describe the behavior of the system on unit level, i.e., each state chart corresponds to exactly one component in the current product variant. This is a typical procedure in model-based testing (MBT), where state charts or similar automaton representations are used to describe component behavior [UL07]. A total of 21 state machines have been modeled for the core product variant *P0*. The state machines are modeled in the same syntax and semantics as described in Chapter 2.1.3.

Example 3.5: BCS State Machine for Core Variant of BCS

Figure 3.6 shows a sample state machine for BCS. It represents the *remote control key* component in the core variant *P0*. The sample state machine consists of three states: RCK_idle, RCK_locking and RCK_unlocking. In addition, four transitions t1 to t4 connect these states. In the sample state machine, the locking and unlocking of the car via the remote control key is modeled. The rck_but_lock and rck_but_unlock events model the pressing of a button. These inputs control the two output events rck_lock and rck_unlock. If no button is pressed or the events have been processed, the state machine is in its idle state RCK_idle.

Lity et al. [LLSG12] defined 15 deltas for BCS to derive state machines for all product variants. Deltas transform state machines into their corresponding variants based on the core variant *P0*. Deltas are able to modify states, transitions and events on different hierarchy levels. The application conditions for the different deltas are boolean formulas over the set of features defined in the BCS feature model. For example, "`ManPW AND CLS AND NOT AS`" is a boolean constraint, which states that a delta is only applied when the manual power window (ManPW) and central locking system (CLS) features are present, but not the alarm system (AS).

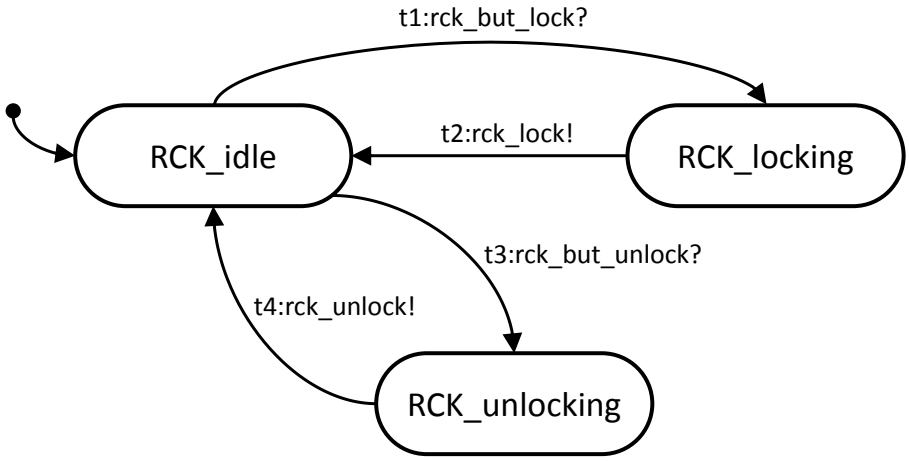

Figure 3.6: State Machine of Remote Control Key in Core Variant (cf. [LLLS12])

Delta-Oriented Architectures. The communication of components and their connections with each other are described as *delta-oriented architectures* [LLL+14] (cf. Chapter 2.3.3). Similar to state machines, a core architecture model has been defined for product variant *P0* [LLLS12]. In context of this thesis, these delta-oriented architecture models are used as *test models* [UL07] for integration testing [Som10] of SPLs as they describe the communication between of components based on their *interfaces*.

Example 3.6: Architecture Model for Core Variant of BCS

Figure 3.6 shows the architecture model of core variant *P0* for BCS. It contains four components: *Human Machine Interface* (HMI), *Manual Power Window* (ManPW), *Finger Protection* (FP) and *Exterior Mirror* (EM). These components are connected via several connectors, labeled with the signals they transfer, e.g., pw_but_up. The HMI component receives input from the environment (represented by no visible sending component). The input is redirected to the other components. Via the HMI, the position of the manual power window and the exterior mirror can be controlled. If a finger is detected, the window can not be moved up any further. Both, manual power window and exterior mirror send their output back into the environment, i.e., to actuators (e.g., motors) not described in the architecture.

For BCS, a total of 25 architectural deltas has been defined to derive all 11,616 possible product variants based on the core architecture [LLLS12]. The deltas defined for BCS only use *add* and *remove* delta operations as *modify*-operations can

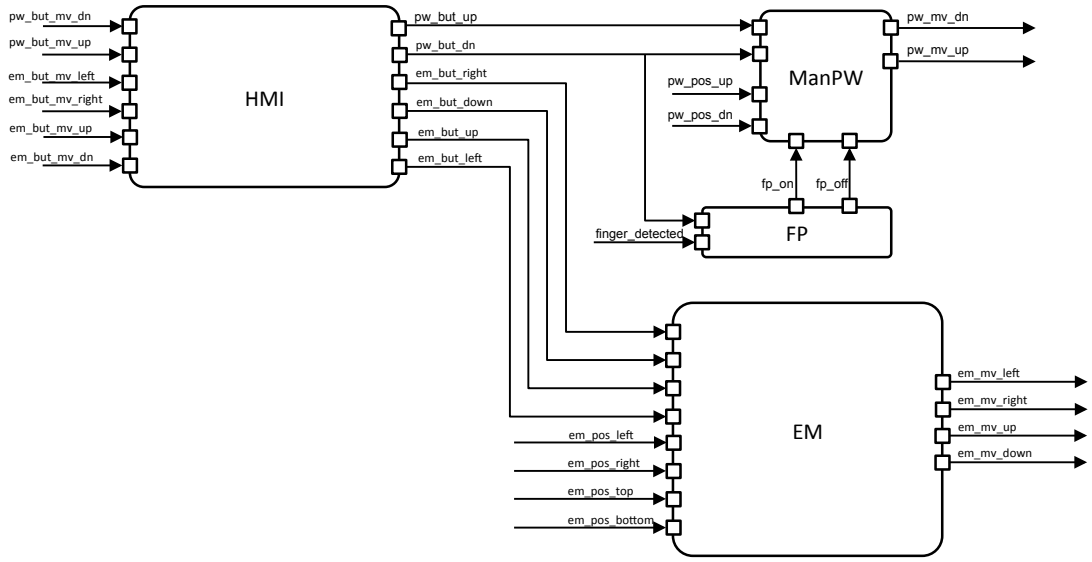

Figure 3.7: Sample Architecture for Core Variant P0 (cf. [LLLS12])

be represented by removing the old variant of the element and adding the modified one (cf. Chapter 2.3.3). Thus, we are able to automatically generate the architectures models for the product variants *P1* to *P17* based on the core architecture.

For BCS, the core architecture and architectural deltas are provided in the DSL Deltarx [LLLS12]. Deltarx supports the automatic generation of product variant architecture models based on the feature model and delta definition. This DSL enables the modeling of SPLs in a textual fashion.

Example 3.7: Example Architectural Delta for BCS

Listing 3.1 shows a sample BCS delta written in the DSL Deltarx [LLLS12]. The delta is designed for the feature *LED Finger Protection* (cf. *when* condition in line 1). It adds functionality to the existing finger protection feature which is present in the core variant. In particular, it adds a new component *LED_FP* to the product variant (cf. lines 7 to 10), which is connected to the *FP* component shown in Figure 3.7 (cf. lines 13 and 14) as well as the environment (cf. lines 15 and 16). For these connectors, two new signals of type boolean are added (cf. lines 3 and 4). They control whether the LED is turned on or off.

Listing 3.1: Delta for LED Finger Protection in DeltaRX

```
1  DLEDFingerProtection when 'LED Finger Protection' {
2    addsignal{
3      led_fp_on boolean
4      led_fp_off boolean
5    }
6
7    addcomponent {
8      LED_FP {
9      }
10   }
11
12   addconnector {
13     fp3(FP,fp_on, fp_on, LED_FP)
14     fp4(FP,fp_off, fp_off, LED_FP)
15     ledfp1(LED_FP,  led_fp_on, led_fp_on, ENV)
16     ledfp2(LED_FP, led_fp_off, led_fp_off, ENV)
17   }
18 }
```

3.2.2 BCS Test Cases and Requirements

BCS provides different artifacts for testing. For testing of software variants, a set of integration test cases has been defined based on the set of delta-oriented architectural test models. In addition, BCS provides requirements and system test cases for system testing, which can be used for regression testing of software versions. BCS does not provide failure information for its variants or versions.

System Requirements. 97 system-level requirements have been defined for BCS [LLLS12]. Each requirement is written in German natural language. Both, *functional* and *non-functional* requirements have been defined to describe the system specification for BCS. The requirements define the specification for *system testing* [Som10]. Requirements are linked to system test cases.

System Test Cases. A set of system test cases have been defined for BCS based on the system requirements. They describe the system's functionality on a user-observable level. System test cases are specifically defined for the overall system, not focusing on specific components or their communication. Thus, system test cases describe tasks to be manually executed by the tester. They resemble typical user tasks to be performed on the finished system. Each test case has a *precondition*, necessary *test steps* and an *expected result*.

Example 3.8: Sample Natural Language Requirement for BCS

The defined requirements are written in natural language, such that a human test expert can interpret them. One sample requirement for the feature *manual power window* is defined as follows:

> "Pressing the Button to close the window leads to the closing of the window as long as the button is pressed."[a]

[a] The original requirements are written in German. This particular requirement is originally defined as: "*Drücken auf den Knopf zum Schließen des Fensters führt dazu, dass das Fenster so lange geschlossen wird, wie der Druck ausgeübt wird.*" [LLLS12]

The precondition defines a state in which the system has to be in to execute the test case. A finite set of test steps define the manually performed actions to execute the test case. The expected result defines the correct behavior of the system. All of these three elements of a system test case are written in natural language for BCS. In total, 128 system-level test cases have been manually defined for BCS based on the system requirements. Each requirement for BCS is covered by at least one system test case, i.e., there exists traceability between test cases and requirements provided in HP Quality Center.

Example 3.9: Sample Natural Language Test Case for BCS

Based on the requirement shown in Example 3.8, the following test case (among others) has been specified for BCS:

> "*Precondition*: The window is not moving. *Step*: The button to move the window up is continuously pressed. *Expected result*: The window moves up."[a]

[a] The original test case is defined as: "*Vorbedingung: Das Fenster steht still. Aktion: Der Knopf zum Schliessen des Fenster wird betaetigt und gehalten. Erwartetes Ergebnis: Das Fenster schliesst sich.*" [LLLS12]

Integration Test Cases. BCS provides integration test cases to test the communication of different components present in a product variant. A total of 92 integration test cases have been defined in previous work for integration testing of BCS [LLLS12, LLAH+16]. They have been specified as MSCs (cf. Chapter 3.1.1). Each MSC describes the communication between at least two components of the system, with only one exception, where a component communicates solely with the environment. The MSCs have been manually designed for the SPL based on the architectural and behavioral test models and are written in a self-defined DSL to be

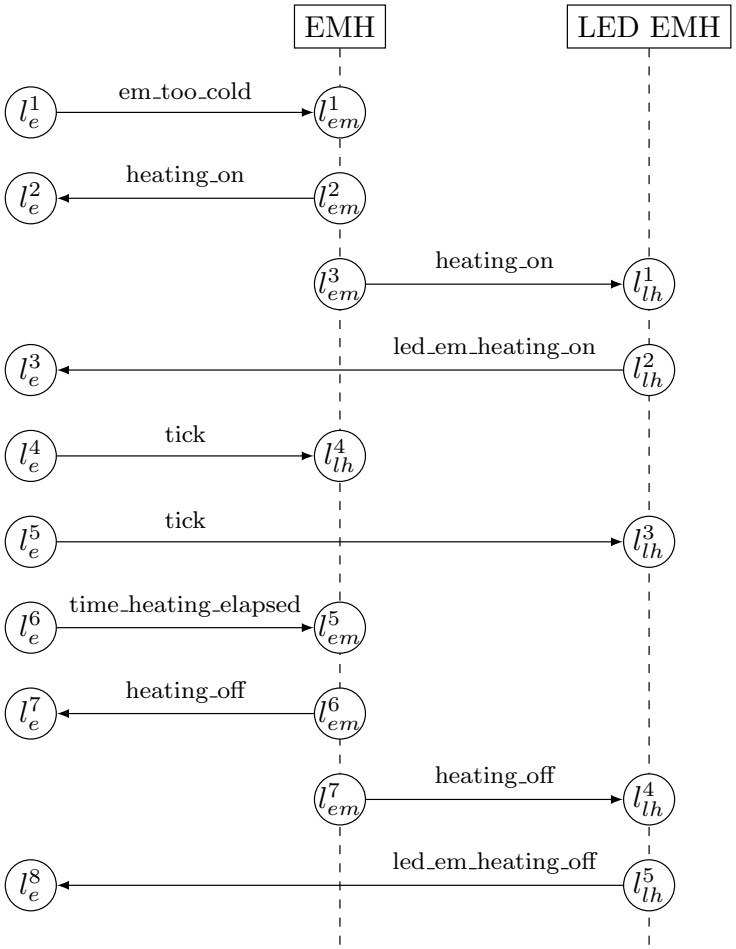

Figure 3.8: Sample MSC Test Case for BCS (cf. [LLLS12])

automatically accessible. As the BCS case study is defined as SPL, not every MSC is applicable to each product variant of the SPL. Their applicability depends on the available components and connectors in the current product variant.

Example 3.10: Sample Message Sequence Chart for BCS

An example for a test case defined as MSC is shown in Figure 3.8. The test case describes an interaction scenario between two components, *Exterior Mirror with heating* (EMH) and a corresponding *LED* (LED EMH). The shown MSC describes how the LED is activated (heating_on) as the temperature is below a certain minimum (em_too_cold). After a certain time (time_heating_elapsed), the heating is turned off again (heating_off).

Failures. As BCS is developed by students in a business game, no failure history is provided. While we do not have explicit failure data, we are able to use experiences from student implementations to seed realistic failures for system testing. In total, we seeded *seven* failures based on failures which actually occurred. We decided for this rather low number of failures, as we do not have enough sufficient data of real failures and restricted the system to a few, but realistic failures.

3.3 Chapter Summary

In this chapter, we laid the foundation of the approaches presented in this thesis. In the first section, we introduced the test artifacts which we use in this thesis. In particular, we described artifacts for testing of software variants and software versions. For testing of product variants, we introduced SPL-related artifacts, namely feature models, features and product variants. As we aim to test individual product variants, we describe test models as generic idea to derive test cases and analyze changes between variants. Deltas are applied to the provided test models to generate test models for specific product variants. We assume that each delta has an application condition based on features, which must be fulfilled to apply the delta and its delta operations. Deltas are able to add, remove and modify parts of the test model. Second, we introduce the artifacts for testing of software versions. Here, we require knowledge about system requirements and test cases. These artifacts are defined in natural language, which is typical for system specifications [SLS11]. We require traceability between these artifacts, i.e., requirements are linked to their corresponding test cases. Additionally, failures have to be reported and linked to their revealing test cases.

In the second section, we introduced the *Body Comfort System* (BCS) case study which is used throughout this thesis as subject system for evaluation. BCS provides artifacts for both, testing of variants and versions. It comprises 27 features which enable a derivation of 11,616 product variants, which are represented as delta-oriented test models. Delta-oriented state machines and architectural test models are provided for BCS, which we use in this thesis to demonstrate the applicability of our black-box testing approaches for software variants. For testing of software versions, BCS provides a set of natural language requirements and test cases. Unfortunately, BCS does not provide realistic failure data to be analyzed.

The following contributions make use of the provided foundations and use BCS as case study for evaluation.

Part II

Black-Box Testing of Software Variants

4 A Model-Based Delta-Oriented Test Case Prioritization Framework

The content of this chapter is mainly based on work published in [LLL+15], [LLAH+16] and [LLS+17].

Contribution

We introduce an extensible framework to prioritize test cases for individual product variants in SPLs. Important test cases are identified in an incremental fashion by exploiting delta information between test models representing product variants. Our test case prioritization framework is extensible and flexible. We provide different instantiations of our testing framework to show its applicability to SPLs.

Testing *software product lines* (SPL) is a difficult task due to the potentially large number of variants and the high redundancy between variants caused by their commonalities [Eng10, MMCDA14]. In this chapter, we propose a regression testing inspired technique to *prioritize test cases* for SPLs. We design our test case prioritization technique as an extensible framework. The framework is able to prioritize test cases based on incremental changes between product variants by stepping from one product variant to the next. Thus, we analyze previously tested product variants and compute important changes which should be tested first. Our test case prioritization framework for SPLs is flexible and can be adapted and extended according to available data and tester's preferences.

First, we define requirements to be fulfilled by a delta-oriented test case prioritization framework. Next, we introduce the framework, which consists of the definition of weight metrics, weight functions, prioritization functions and an framework instantiation phase. Afterwards, we propose sample instantiations based structural and behavioral test models. They are based on delta-oriented architectures and delta-oriented state machines, which we described in Chapter 3.1.1. To show the novelty of our testing framework, we evaluate it using the BCS case study (cf. Chapter 3.2). Afterwards, we explain and compare related work in the domain of SPL testing. Finally, the chapter is concluded providing a summary and a discussion of potential future work.

4.1 Framework Requirements

To provide a framework for delta-oriented test case prioritization we first have to establish a specification in terms of necessary requirements to be fulfilled. In the following, we define the *functional* and *non-functional* requirements to be fulfilled by our testing framework.

4.1.1 Definition of Functional Requirements

We require that certain functionalities have to be provided for the purpose of our test case prioritization framework for SPLs, which is a reduction of testing effort in SPLs. In general, the testing framework shall support the prioritization of test cases for individual product variants in an SPL. The prioritization shall be performed based on already tested product variants and their changes between the currently tested product variant. We define three *functional requirements* for our testing framework [LLS+17].

FR1: Analysis of Delta-Oriented Test Models. We require our testing framework to be able to handle *delta-oriented SPLs* as input. Furthermore, we require it to be able to handle delta-oriented test models, i.e., it shall support model-based testing as we assume to have no knowledge about source code. Thus, the delta-oriented test models represent the specification of the system, based on which we are able to make assumptions about changes between variants. The framework supports different types of test models based on our abstract test model notation introduced in Chapter 3.1.1. For example, in integration testing, architecture models specify the communication of components in a software system. Changes between product variants are identified via the deltas provided for the test models.

FR2: Definition of Weight Metrics for Test Model Elements. The framework has to be able to compute the *priority* of test cases based on delta-oriented test models,. To this end, we require the framework to be able to derive *weights* for certain model elements. For example, for architectural models we require the framework to be able to compute *component weights*, i.e., each component is analyzed separately and given a certain weight value. Weights are computed using a set of user-definable *weight metrics* which are aggregated in a *weight function*. Weights shall be definable based on the analysis of the provided test models, deltas and additional data available in black-box testing, e.g., test case history or risk factors.

FR3: Flexible Definition of Test Case Prioritization Functions. Based on the weight computation, the framework shall support the definition of *test case prioritization functions*. These functions determine the priority of a test case. Thus, the results are ordered lists of test cases, which are executed in their ordering to maximize testing effectiveness.

Of course, the definition of priority is crucial for the definition of test case prioritization functions. Here, no restrictions are made on how to define which test cases are more important than others. However, the prioritization functions have to be defined based on the weight functions, which combine a user-specific set of weight metrics. Thus, an abstraction from the used weight metrics can be made. We require the framework to be adjustable in the sense, that the defined weight metrics can be further adjusted in the final test case prioritization using *weighting factors*, i.e., the impact of each weight metric is user-specific.

Test cases have to be defined according to the provided test models. Consequently, test cases to be prioritized have to relate to the test model contents. Each test case describes a test scenario. The SPL framework does not focus on the generation or creation of test cases. It only prioritizes test cases, if they are applicable for a certain product variant, i.e., if their tested behavior is actually present in the current product variant under test (PUT).

4.1.2 Definition of Non-Functional Requirements

Besides the functional requirements, our framework for delta-oriented test case prioritization shall also fulfill three *non-functional* properties.

NFR1: Extensibility. A major aspect of our test case prioritization framework is the flexibility to be extended by the user according to available data. In particular, we require the framework to be easily extensible concerning the supported weight metrics. Additional information might be available to the user, e.g., different test models or different types of meta-data. Thus, we require that the weight computation is extensible to support new weight metrics. Additionally, the framework shall be able to support different types of prioritization functions which can be defined by the user, e.g., to focus on test case dissimilarity. This flexibility leads to a framework which supports different types of data and, thus, different aspects of testing.

NFR2: Controllability. After selecting a set of weight metrics and creating a framework instance, we require the user to be able to further adapt and control the test case prioritization. For this, no new instance shall be necessary. Rather, using different *weighting factors*, the user shall be able to adjust the test case prioritization to his needs. This supports regression testing, where a certain aspect or weight metric might gain importance over time. For example, the overall coverage of the system is more important in the beginning of a project than focusing on changes between variants, i.e., dissimilarity-based test case prioritization aspects might be more important than delta-oriented ones.

NFR3: Usability. Manual test case prioritization is a difficult task, especially for large-scale SPLs. Right now, only custom-tailored software exists to solve these issues, as no other frameworks exist to prioritize test cases incrementally for a set

of product variants. This leads to the non-functional requirement to support *usability*, i.e., our testing framework shall support the user in the instance creation and resulting test case prioritization approaches shall be faster deployed than a custom-tailored software. In particular, we require extensions to be easily integrable. On the other side, the framework shall produce results comparable to custom-tailored approaches [LLL+15, LLAH+16]. This enables the goal of a customer-adaptable test case prioritization to increase testing efficiency, due to less effort in the integration of test case prioritization, while being effective, i.e., important test cases are prioritized and, thus, executed first.

4.2 Framework Definition

General Approach. Based on the previously defined functional and non-functional requirements we define a delta-oriented test case prioritization framework for SPLs. The framework is designed to cope with a set of input artifacts, namely test cases, delta-oriented test models and regression deltas (cf. left-hand side of Figure 4.1). Each product variant is represented by a test model. Based on the input data, we are able to define weight metrics, weight functions and test case prioritization functions (cf. right-hand side of Figure 4.1). The framework aims to identify important test cases according to changes in product variants that have to be retested. Our test case prioritization is performed incrementally for one product variant at the time. For each product variant, the resulting test case prioritization functions compute priority value for each *reusable* test case, i.e., for test cases which are applicable to the current product variant and have been executed at least once in before. New test cases, which are first applied for a product variant, are not prioritized. We assume that that these test cases have always to be executed to ensure that the new functionality is correctly implemented. The ingredients of the framework and potential instantiations are explained in the following.

4.2.1 Definition of Input Artifacts

Test Case Definition. *Test cases* are defined for a certain product variant in the SPL. The framework does not focus on the design or generation of test cases. We simply assume that a mechanism has been implemented such that for each product variant a set of new test cases $TC_{new} \subseteq \mathcal{TC}$ can be assigned when necessary. In other words, we only require new test cases for a product variant p_i if a certain functionality has never been tested before in previous product variants under test $P_{tested} = \{p_1, \ldots, p_n\}$. By definition, the framework is able to cope with a wide variety of test case types due to its generic nature, as long as they correspond to the provided test models [UL07].

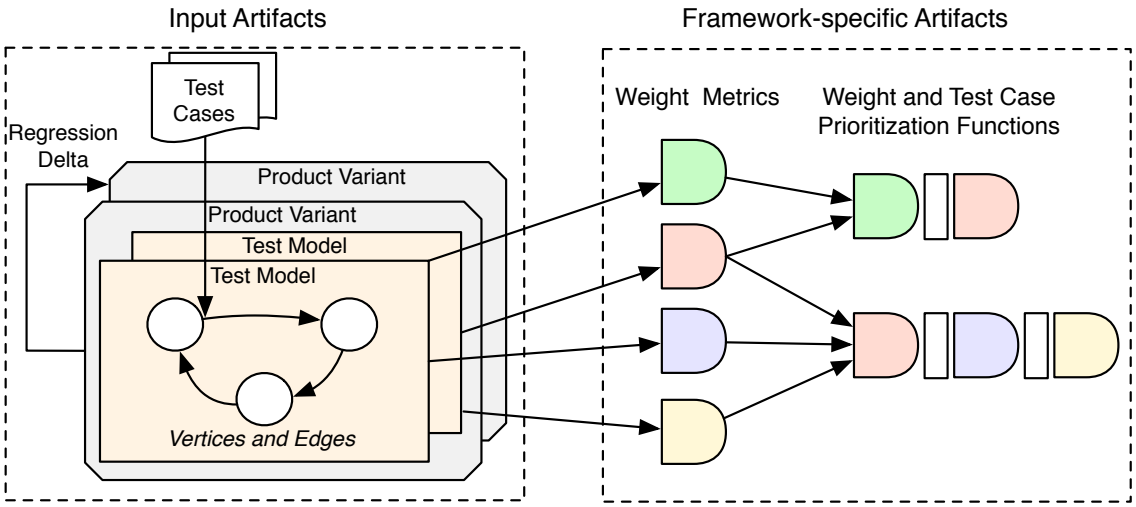

Figure 4.1: SPL Test Case Prioritization Framework Artifacts

Test Model Definition. Each product variant of the SPL is specified by a set of *test models* (cf. left-hand side of Figure 4.1). The framework supports different types of test models, as long as they are specified as described in Chapter 3.1.1, i.e., they contain a set of vertices and edges. Test cases cover test model element, i.e., each test case $tc \in \mathcal{TC}$ covers a set of vertices and edges defined as $\Omega_{tc} \subseteq \Omega$. To ensure test coverage, we require that enough test cases are provided such that each vertex and edge of a product variant's test model is covered by at least one test case applicable to that particular product variant. Consequently, we require that

$$\forall\, \omega \in \Omega\ \exists\, tc \in \mathcal{TC} : \omega \in \Omega_{tc} \wedge (tc \in \mathcal{TC}_{new} \vee tc \in \mathcal{TC}_{reusable})$$

for a each product variant under test holds. In addition, we require knowledge about the deltas, i.e., the transformations for the core variant to identify changes between variants.

Regression Deltas. Each product variant is part of the same SPL, i.e., there exists a relationship between these variants in terms of their commonalities and variability. Based on the available deltas, we are able to compute a *regression delta* [LSKL12] between two arbitrary product variants of the SPL, describing the transformational differences between them (cf. Chapter 3.1.1). These regression deltas are key to the identification of important changes in the a currently tested product variant and, thus, vital to the framework. Their computation is based on the symmetrical difference of deltas defined for the core variant. This aspect is described in the next subsection.

4.2.2 First Applied Regression Delta Computation

The framework supports the analysis of previously untested parts of a PUT. This is important as the framework focuses on the identification of important changes and, based on this, derives priority values for the test cases applicable to the current PUT.

To this end, we analyze the regression deltas between the current PUT p_i and all previously tested variants P_{tested}. Constructing the intersection of regression deltas leads to deltas, which have never been applied before as the contained delta operations are present in all regression deltas for product variants in P_{tested}. In other words, the intersection of regression deltas leads to a difference which has to be applied to all previously tested variants, indicating new product variant specific elements to be tested. Thus, they allow for the identification of never before tested parts of the PUT. Formally, we compute these *first applied regression deltas* as defined in Definition 4.1.

Definition 4.1: First Applied Regression Delta Computation

Let $P_{tested} \subseteq P_{SPL}$ be a non-empty set of previously tested product variants, $p_{current}$ the current PUT, $\Delta_{p,p_{current}}$ the set of regression deltas between a product variant $p \in P_{tested}$ and $p_{current}$ such that $p_{current} \notin P_{tested}$.

We define the set of *first applied regression deltas* $\Delta^{new}_{p_{current}} \subseteq \Delta_{SPL}$ as the intersection of delta operations ($\bigcap$) in regression deltas as follows:

$$\Delta^{new}_{p_{current}} = \bigcap_{j=1}^{|P_{tested}|} \delta_{p_j,p_{current}}$$

4.2.3 Definition of Weight Metrics and Functions

Intention. One fundamental aspect of the test case prioritization framework is the computation of changes in an abstract test model compared to previously tested product variants and their test model representation. Different aspects might be of importance to identify the impact of changes between different variants under test. Thus, the framework supports the definition of distinct *weight metrics*. These metrics resemble the degree of changes a certain part of the system has undergone. A higher degree of change results in a higher weight, indicating significance for retesting compared to parts with lower weights.

Available Information. Due to the focus on model-based testing for SPLs, the framework is designed to support the computation of *model element weights*,

which depend on type of available test models. Independently of the applied weight metrics, we assume that the underlying information is based on black-box data, i.e., no source code access is available for the PUT.

Weight Metric Structure. As a result, a set of weight metrics $\mathcal{M} = \{m_1, \ldots, m_n\}$ is defined for test case prioritization. Each weight metric m computes a distinct weight value based on available test artifacts for the current PUT. To support SPL testing, we advise to define weight metrics based on differences between product variants, such that the resulting weight values reflect how much the specification of one product variant differs from the previously tested product variants $P_{tested} \subseteq P_{SPL}$.

For model-based testing, we define weight metrics for model elements $\omega \in \Omega$ (cf. Chapter 3.1.1) to be of the signature $m : \Omega \to \mathbb{R}$, i.e., they are applied to vertices or edges of a test model. To support the comparison and combination of metrics, their computation should be normalized by design, e.g., by ensuring that $m : \Omega \to [0, 1]$ holds. Weight metrics should be always applied to elements of the respective type (i.e., vertices or edges) of the product variant to identify parts of the test model which have changed more than others.

As this chapter focuses on incremental delta-oriented testing, the weight computation is performed for each PUT anew. In particular, the observed new changes in the current PUT depend on the ordering of product variants, the number of previously tested product variants and the relevant deltas. The first applied delta application (cf. Chapter 4.2.2) supports the identification of changed parts in the current PUT.

Weight Functions. Based on different weight metrics, the user is able to define a set of *weight functions* $w : \Omega \to \mathbb{R}$ which are applied to a test model element $\omega \in \Omega$ (cf. Chapter 3.1.1). A weight function consists of one or more weight metrics, whose results are combined. Aggregating different weight metrics is useful to take different aspects of testing into account simultaneously. For example, the changes between incoming edges might be of importance as well as the changes between outgoing edges of a vertex in a test model, as they indicate different types of new behavior. Thus, two corresponding weight metrics are defined and combined in one weight function. However, it might be the case that one weight metric is more important than the other. Thus, our test case prioritization framework for SPLs includes the definition of *weighting factors* $A = \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathbb{R}$ where $1 = \sum_{m=1}^{|A|} \alpha_m$ holds. Using the weighting factors, the influence of single weight metrics can be adjusted by the user or stakeholder according to the current state of development and testing.

Definition 4.2: Weight Functions

Let $A = \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathbb{R}$ be the set of weight factors such that $1 = \sum_{i=1}^{n} \alpha_i$ holds, $M_{selected} \subseteq \mathcal{M}$ the set of selected weight metrics and Ω be the set of test model elements.

We define the *structure for weight functions* $w : \Omega \to \mathbb{R}$ for a test model element $\omega \in \Omega$ to be as follows:

$$w(\omega) = \alpha_1 \cdot m_1 + \alpha_2 \cdot m_2 + \ldots + \alpha_n \cdot m_n$$

Guidelines for Weight Metrics. Generally, the test case prioritization framework does not restrict the definition of weight metrics or functions. In particular, arbitrary weight metrics $m \in \mathcal{M}$ can be combined freely. The test case prioritization is very flexible due to the weighting factors, which allow for individual adjustments of weight metric influences. However, we provide some general guidelines when designing weight metrics.

The framework is designed for incremental testing, i.e., the defined weight metrics should adapt to this concept and take changes between product variants into account. Otherwise, the nature of SPLs in terms of their commonalities is not exploited and the reuse potential of test cases is reduced. To this end, we suggest to use the available delta-information to derive weight metrics for each PUT anew.

Example 4.1: Architectural Weight Metrics

Assume that weight values for each component $c \in C$ of a architecture test model are to be computed. Thus, a first weight metric is defined based on the delta-changes of incoming connectors $IC_c \in CON$ of the component's interface. In case that an incoming connector has been added or removed by a delta for the first time, we assume that the component has changed and, thus, should be tested again. The metric is normalized using the set of overall incoming connectors $I_c \in CON$, such that $m_{in}(c) = \frac{|IC_c|}{|I_c|}$. Now we are able to combine this weight metrics with others using weighting factors. For example, we analogously define $m_{out}(c)$ for changed outgoing connectors of component c. Using two weighting factors, the resulting weight function is defined as $w(c) = \alpha \cdot m_{in}(c) + \beta \cdot m_{out}(c), \alpha + \beta = 1$. Applying this function to each component c of the PUT results in separate weight values between 0 and 1, allowing to distinguish the degree of changes between variants. Results are influenced by changing the weight factors to favor different weight metrics.

4.2.4 Definition of Test Case Prioritization Functions

Test Case Prioritization Functions. The aim of the framework is the prioritization of test cases for product variants of delta-oriented SPLs. Thus, it supports the definition of *test case prioritization functions* which assign a priority value to each test case applicable to the current PUT. The higher the priority value, the earlier a test case is executed. We do not prioritize new or obsolete test cases for a current PUT, as new test cases have to always be executed and obsolete test cases are not executable. Instead, we focus our prioritization on reusable test cases, which have been executed for a previously tested product variant. Consequently, to prioritize test cases we order them in descending fashion according to their priority values.

Test Case Priority. As a result of applying the defined test case prioritization functions, each test case is assigned with a priority value according to their covered test model elements. The higher the number of covered changes, the higher the importance of a test case. In certain circumstances the weight of a vertex might be 0, i.e., no changes occurred. Test cases, which only refer to these unchanged test model elements should also receive a priority value of 0. These test cases do not have to be executed again, i.e., they are reusable, but not retestable. These test cases still can be executed after all other test cases with a priority > 0 have been performed. Thus, our testing framework for SPLs also supports test case selection.

Prioritization Computation. A test case prioritization function has the signature $prio : \mathcal{TC} \times P_{SPL} \rightarrow \mathbb{R}$. If a test case corresponds to a changed or influenced part of the current PUT p, for which the prioritization function is applied, the resulting priority value should be greater than 0. To this end, the covered test model elements $\Omega_{tc} \subseteq \Omega$ of a test case are of relevance. These mappings allow for a detailed analysis of the test cases and a priority computation for each applicable test case for a PUT. As a result, we are able to map the weight values derived by the weight functions to the test cases.

Example 4.2: Component-based Test Case Prioritization Function

Let $w : C \rightarrow \mathbb{R}$ be a weight function for a component $c \in C$. The detailed weight computation is not of relevance. Let $TC_{reuse_p} \subseteq \mathcal{TC}$ be the set of test cases reusable for a product variant $p \in P_{SPL}$. A sample test case prioritization function is defined based on the components C_{tc} covered by a test case $tc \in \mathcal{TC}$. It sums the weights of each distinct covered component $c \in C_{tc}$ up to compute the test case priority. We define the test case prioritization function $prio_{comp} : \mathcal{TC} \times P_{SPL} \rightarrow \mathbb{R}$ for a test case $tc \in TC_{reuse_p}$ as:

$$prio_{comp}(tc, p) = \sum_{j=1}^{n} \{w(c_j) \mid c_j \in C_{tc}, n = |C_{tc}|\}$$

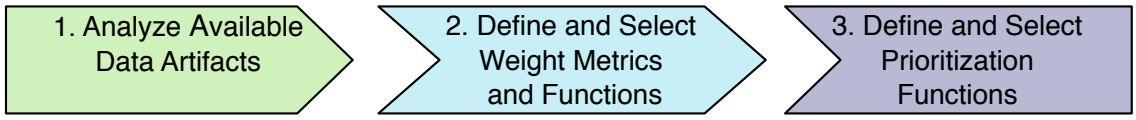

Figure 4.2: Guideline Steps for Framework Instantiation (cf. [LLS+17])

4.2.5 Guidelines for Framework Instantiation and Configuration

To guide the user in the framework instantiation, we distinguish three basic steps to instantiate the framework as displayed in Figure 4.2.

1. Analyze Available Data. First, the available data has to be analyzed in order to assess which weight metrics can be defined. The framework is based on the assumption that test models derived in a model-based testing process (cf. Chapter 2.1.3) are available for testing. Thus, one major aspect is the type of available test models, e.g., if only structural models are available or if additional behavioral models can be used for test case prioritization. Test models have to be automatically processable to be of use for the framework, i.e., they have to be available in a structured form, e.g., provided in a *domain-specific language* (DSL) [Sel07].

In addition to the test models, the framework is designed for incremental testing. Hence, it requires a set of product variants to be tested and their ordering. These variants have to be mapped to available artifacts, e.g., using feature configurations. In case deltas are available, the test models for product variants can be generated automatically. In addition, the framework can make use of deltas to compute novel regression deltas, which help to identify changed parts (cf. Chapter 4.2.2). However, if no explicit delta knowledge is available, the change information can be automatically derived, e.g., using model differencing techniques or extractive delta module generation [PKK+15, WRSS17]. A set of test cases has to be provided for the SPL, which are automatically categorized for product variants. For each product variant, based on their test models, the set of applicable test cases is computed.

2. Define and Select Weight Metrics and Functions. As explained in earlier subsections, the framework is based around the concepts of weight computations (cf. Chapters 4.2.3 and 4.2.4). Thus, based on the available data, the user has to define a set of weight metrics as foundation for the framework instantiation. These metrics specify the focus of testing and which data is accessed, e.g., by performing structural analyses of changes between test models. Weight metrics can be aggregated to weight functions using weight factors. The user has to define and select the weight metrics which shall be available in the framework instance.

Using the available weight metrics, weight functions are constructed. The test engineer has to specify which metrics shall be aggregated. Using weighting factors,

the influence of weight metrics can be adjusted after the instantiation. Of course, using a weight of 0 allows the tester to omit certain weight metrics from the final result, providing a selection of weight metrics.

3. Define and Select Prioritization Functions. Based on weight metrics and functions defined in the previous step, test case prioritization functions are defined. They are interchangeable and allow the prioritization of test cases in different ways, i.e., users are able to perform test case prioritization functions separately on the same set of test cases and product variants. Of course, the selected test case prioritization functions have to match the available data, i.e., they can only be applied if the required input artifacts are available for testing (cf. Step 1).

Once the test case prioritization functions have been defined and selected, the test case prioritization can be performed. If implemented correctly, the execution can be performed in a fully autonomous fashion, i.e., no additional user input is required. The only user-interaction is performed when configuring the test case prioritization, i.e., setting the weight factors and selecting the product variants to be tested. The results of a test case prioritization are priority values for each test case applicable to the analyzed product variants. The framework only prioritizes reusable test cases, i.e., only test cases which are applicable to a current product variant and which have been applicable to previous variants are prioritized. As a result, each test case $tc \in TC_{reuse}$ receives a priority value. Using these priorities, test cases are executed in descending order, beginning from the highest value. In case that two distinct test cases receive the same value, the ordering is random as both are of the same importance. Testing of prioritized test cases should continue until available testing resources are exhausted.

4.3 Framework Implementation

Implementation in Eclipse. Using the defined guidelines, we implemented instances of our testing framework in Java. In particular, we provide the architecture test models using the EMF-Framework[1] and by defining a DSL using the Xtext and Xtend frameworks[2] for Eclipse [3]. A DSL allows the user to write a grammar for a particular need [VBD+13]. In case of this thesis, we specifically define DSLs for the definition of architecture test models and message sequence charts, which represent test cases. Their definitions are linked via grammar-mixin, i.e., the test case grammar references the architecture descriptions to ensure the validity of test cases.

[1] Eclipse Modeling Framework, website: `https://www.eclipse.org/modeling/emf/`, date: January 4th, 2017

[2] Xtext and Xtend frameworks [Bet13], websites: `https://eclipse.org/Xtext/` and `https://eclipse.org/xtend/`, date: January 4th, 2017

[3] Eclipse IDE for Java, website: `https://eclipse.org/ide/`, date: January 4th, 2017

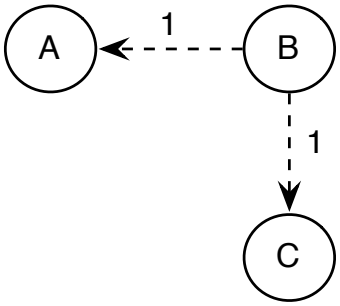

Figure 4.3: Sample Delta Graph (cf. [LLL+15])

In addition, the architectural DSL DELTARX supports the definition of deltas, which can be applied automatically given a valid feature configuration [LLLS12, LLL+14]. We are able to directly integrate the EMF-based DSL editors into ECLIPSE based on the IDE's plugin-based structure. Thus, ECLIPSE is the foundation for the technical realization of framework instantiations.

Delta Graphs. Novel regression deltas are described as directed and weighted *delta graphs* [LLL+15]. Delta graph vertices represent vertices of the current PUT's test model. Edges in these graphs represent the degree of changes in the test model edges. The weight of each delta graph edge specifies the number of changes between two particular vertices which occurred for the first time in the current product variant. While such a graph representation is not required for the execution of the test case prioritization, it helps to understand the changes between variants in an easy to grasp, graphical fashion.

Example 4.3: Sample Delta Graph

An example delta graph is shown in Figure 4.3. In this particular graph, three components A, B and C are shown. Between A and B, one connector has been changed as well as between B and C, indicated by the dotted edges with a value of 1. In terms of delta graphs, it does not matter if a change was an *add* or *remove*, but only that the interface has changed and in which direction, indicated by the directed edges.

Provided Weight Metrics. We implemented our testing framework to prioritize integration test cases for SPLs. Thus, we use architectural test models as input to prioritize test cases defined as message sequence charts. We further provide behavioral test models for a more fine-granular analysis of changes. In the following, we explain the weight metrics we implemented as foundation for our testing framework based on the available artifacts. We distinguish between architecture-based and behavior-based weight metrics as foundation for our evaluation.

4.3.1 Architecture-Based Weight Metrics

We first define a set of weight metrics which assess the weight of components based on the architecture models provided for each product variant. We aim to prioritize integration test cases using these models and concepts.

Component Interface Changes. For each component $c \in C$, we are able to analyze its interface, i.e., which incoming and outgoing connectors the component has and to what degree they have changed. In particular, we are interested in changes which occur for the first time in the current PUT, i.e., the underlying delta operations have not been applied to previously tested product variants P_{tested}. The set of changed incoming connectors $IC_c \subseteq CON$ of component $c \in C$ are of interest as well as the set of changed outgoing connectors $OC_c \subseteq CON$. Using the novel regression delta computation shown in Chapter 4.2.2 in Definition 4.1, we are able to detect interface changes which occur for the very first time. We normalize these changed connectors by the number of currently present connectors in the component, i.e., its set of incoming connectors $I_c \subseteq CON$ and outgoing connectors $O_c \subseteq CON$. This leads to the two interface weights metrics, for incoming and outgoing interface changes, respectively.

Definition 4.3: Incoming Interface Change Weight Metrics

Let $I_c \subseteq CON$ be the set of incoming connectors of component $c \in C$ and $IC_c \subseteq I_c$ the set of changed incoming connectors of component $c \in C$.

We define the incoming interface weight metric $m_{in} : C \to \mathbb{R}$ for a component $c \in C$ as:

$$m_{in}(c) = \frac{|IC_c|}{|I_c|}$$

Definition 4.4: Outgoing Interface Change Weight Metrics

Let $O_c \subseteq CON$ bw the set of outgoing connectors of component $c \in C$ and $OC_c \subseteq O_c$ the set of changed outgoing connectors of component $c \in C$.

We define the outgoing interface weight metric $m_{out} : C \to \mathbb{R}$ for a component $c \in C$ as:

$$m_{out}(c) = \frac{|OC_c|}{|O_c|}$$

Multi Product Deltas. We introduced the concept of *multi product deltas* (MPD) in Chapter 3.1.1. They describe changes in product variants, which are not detectable by analyzing first applied regression deltas alone. This is due the fact, that all deltas related to a test model element have occurred in isolation in previous product variants, but never in a particular combination. Thus, it is important to analyze the occurrences of test model elements in all previously tested product variants to identify MPDs. We define a *multi product delta weight metric* $m_{MPD} : C \to \mathbb{R}$ based on the degree of MPDs of a component c. In particular, we measure how many differences the current interface has to the most similar previously tested variant of the interface, i.e., the number of differing connectors. To normalize the results, we count the number of previously tested product variants in which the test model element occurred. The higher the number of occurrences the lower is the impact of the MPD. We define this set as $P_c \subseteq P_{tested}$. Based on both, the MPDs of a test model element ω and the number of its occurrences, we define an MPD weight metric as specified in Definition 4.5.

Definition 4.5: MPD Weight Metric

Let MPD_c the number of multi product deltas for component $c \in C$ and $P_c \subseteq P_{tested}$ the set of tested product variants containing component c.

We define the *MPD weight metric* $m_{MPD} : C \to \mathbb{R}$ for a component $c \in C$ as:

$$m_{MPD}(c) = \frac{MPD_c}{|P_c|}$$

4.3.2 Behavior-Based Weight Metrics

While the architecture-based weight metrics can be derived from structural data, i.e., architecture test models and delta information, we implement additional weight metrics based on *behavioral changes* between product variants [LLAH+16]. These metrics require *delta-oriented state machines* (cf. Chapter 2.3.3), or similar types of behavioral models, to define the behavioral specification of each component in the current PUT. Different levels of test models (e.g., structural and behavioral) allow for a more fine-granular analysis of changes between product variants. We define three behavioral weight metrics based on changes in state machines: *behavioral component weights*, *direct signal weights* and *indirect signal weights*. While behavioral MPDs could be analyzed as well, we do not introduce such a weight metric, as we use these metrics to test integration testing. In other contexts, behavioral MPDs are useful and can be provided based on our testing framework's definitions.

Behavioral Component Weight. Based on deltas, we define a weight metric based on behavioral changes of a state machine of a component. To this end, we measure the number of changed states $CS_c \subseteq S_c$ and *changed transitions* $TS_c \subseteq T_c$ in the state machine of component $c \in C$. Using the available delta information, we are able to extract this knowledge directly from the deltas $\delta^{SM} \in \delta^{SM}_{SPL}$ which correspond to component c. These weight metrics can be normalized using the number of states S_c and transitions T_c in component c, resulting in the following two behavioral weight metrics.

Definition 4.6: State Change Weight Metrics

Let $S_c \subseteq S$ be the set of states in the state machine of a component $c \in C$ and $CS_c \subseteq S_c$ the set of changed states for component $c \in C$.

We define the state change weight metric $m_{state} : C \to \mathbb{R}$ which measures the behavioral component weight of a component $c \in C$ based on its changed states as follows:

$$m_{state}(c) = \frac{|CS_c|}{|S_c|}$$

Definition 4.7: Transition Change Weight Metrics

Let $T_c \subseteq T$ be the set of transitions in the state machine defined for component $c \in C$ and $TS_c \subseteq T_c$ the set of changed transitions for component $c \in C$.

We define the transition change weight metric $m_{tran} : C \to \mathbb{R}$ which measures the behavioral component weight of a component $c \in C$ based on its changed transitions as follows:

$$m_{tran}(c) = \frac{|TS_c|}{|T_c|}$$

Direct Signal Weights. Besides the changes of the state machine itself, changes of the state machine can also influence the signals send by a component $c \in C$. This is due the fact, that input events $\Sigma_I \in \Sigma_c$ and output events $\Sigma_O \in \Sigma_c$ correlate to input signals $\Pi_I \in \Pi_c$ and output signals $\Pi_O \in \Pi_c$ of the component, respectively. Thus, an internal change that occurs within the state machine can influence the communication on architectural level. To this end, we define a *direct signal weight* metric, i.e., we measure the number of deltas $\Delta_t \subseteq \Delta^{SM}_{new}$ which *directly* influence transitions and, thus, events which correspond to signals. A direct influence is

observed if a delta operation adds, removes or modifies a transition that either uses an incoming event as trigger or broadcasts one or more outgoing events. As internal τ-events (cf. Chapter 2.1.3) are only visible within the state machine and do not correspond to any signals on interface level, we do not compute any direct signal weights for internal events. For the direct signal weight metric, we assess the quantity of these deltas for each signal $\pi \in \Pi$ of the current architecture. To this end, we define a function $changes_\pi : \Pi \times \mathcal{P}(\Delta_{SPL}^{SM}) \to \mathbb{N}$, which simply counts these change occurrences in behavioral deltas.

We normalize the number of changes for of a certain signal by the number of all signals received or send by a component $c \in C$, denoted as Π_c. This leads to a *direct signal weight metric* for a signal $\pi \in \Pi_c$ for component $c \in C$ in PUT $p \in P_{SPL}$ based on the set of deltas, which contain operations related to a transition t, referred to as Δ_t. We define the weight metric in Definition 4.8 [LLAH+16].

Definition 4.8: Direct Signal Weight Metric

Let $C_p \subseteq C$ be the set of components in product variant $p \in P_{SPL}$, $\Pi_c \subseteq \Pi$ the set of signals sent or received by component $c \in C$, $\Delta_t \subseteq \Delta_{SPL}^{\mathcal{SM}}$ the set of transition-related deltas and $changes_\pi : \Pi \times \mathcal{P}(\Delta_{SPL}^{SM}) \to \mathbb{N}$ the function which returns the number of change occurrences in a set of behavioral deltas.

We define the *signal weight metric* $m_{direct_\pi} : C \times \Pi \to \mathbb{R}$, which measures the direct signal weight of a signal $\pi \in \Pi_p$ in component $c \in C_p$ as follows:

$$m_{direct_\pi}(c, \pi) = \frac{changes_\pi(\pi, \Delta_t)}{|\Pi_c|}$$

The higher the degree of changes for a certain signal within a component is, the higher is the likelihood that these changes influence the overall communication in unprecedented ways. Thus, this weight metric supports the integration testing of SPLs if behavioral knowledge is available.

Indirect Signal Weights. Besides the direct influence of deltas in state machines, there also exist *indirect* influences of performed changes to indirectly influenced events within a component [LLAH+16]. These indirect changes are not as easily detected as direct ones as they are only observable when analyzing paths within a state machine. We define a path as a set of transitions and states which can be traversed one after another. We argue, that an outgoing signal of a component is indirectly influenced, if there exists a path from a directly influenced transition to a transition, which sends the particular outgoing signal. Thus, indirect weights require a more sophisticated analysis compared to direct signal weights. We use an approach which is similar to slicing techniques used for change impact analy-

sis [ACH+13]. Our pseudo-algorithm is shown in Algorithm 1.

The analysis starts with a transition that is directly influenced by a delta $\delta \in \Delta_t$ and for which the trigger corresponds to an incoming signal of the component, i.e., $\sigma \in \Sigma_I$ (cf. Line 2 in Algorithm 1). In other words, the analysis starts with transitions that are fired by other components and have been changed compared to previously tested product variants. For such a transition t, we analyze and store the broadcast events $\Sigma_B \subseteq \Sigma$ as current events $\Sigma_{current} \subseteq \Sigma$ (cf. Line 5 in Algorithm 1). Our analysis is performed recursively, starting with current transition t (cf. line 6 and lines 10 to 29 in Algorithm 1). We follow this transition, analyzing potential paths following its triggering. Thus, the transition is fired, i.e., the target state of the transition is the new current state $s_{current} \in S$ (cf. line 11 in Algorithm 1). For all of the outgoing transitions t_{out} of $s_{current}$, we check if they can be triggered by the events stored in $\Sigma_{current}$ and have not been visited yet (cf. line 13 in Algorithm 1). In case such a transition has been found, a new "branch" of the path is started, i.e., similar to the steps explained previously, we traverse the transitions and store the events they broadcast. The events used to trigger transitions to move along the path are removed from the current events as they are only visible in one step of the system (cf. lines 12 and 16 in Algorithm 1). To avoid loops in the analysis, we memorize all transitions which have already been traversed, which is similar to a slicing stop criterion [ACH+13] (cf. lines 4 and 14 in Algorithm 1). Another stop criterion of the path analysis are transitions, which broadcast events that correspond to outgoing signals, i.e., $\exists\, \sigma_o : \sigma_o \in \Sigma_O \wedge \sigma_o \in \Sigma_B$ (cf. line 18 in Algorithm 1). These types of events (and signals) are the ones that are indirectly influenced by the original transition t and, thus, the analysis does not continue after such an event has been found. The approach stores these events, as they represent our result set.

For the indirect signal weight metric, we are interested in events which are in the set of broadcast events for the traversed transitions and correspond to signals on architectural level, i.e., they represent an output used for communication. These events are indirectly influenced by the original change in transition t, which was the starting point of the path. Formally, we focus on signal pairs $influence_c \subseteq I_c \times O_c$ in a component c, where an incoming signal influences the outgoing signal [LLAH+16]. Once the path has been fully traversed, we count the number of occurrences of events $\Sigma_{B_t} influenced$ corresponding to a certain outgoing signal in the path (cf. Line 26 in Algorithm 1). To this end, we define a function $impact : \Pi \times \mathcal{P}(CON \times CON) \to \mathbb{N}$, which counts the occurrences of an event σ_π correlating to signal π in the analyzed path. We normalize these findings by the number of overall influences in component c. Consequently, we define a *indirect signal weight metric* $m_{indirect} : C \times \Pi \to \mathbb{R}$ in Definition 4.9 [LLAH+16]. In addition, we show an example for our behavioral weight metrics based on the changes of state machines in Example 4.4.

Algorithm 1 Path Analysis to find Indirectly Influenced Events/Signals

1: **procedure** MainAlgorithm(Δ_t, SM)
2: $\Sigma_{influenced} \leftarrow \emptyset$ ▷ Initialize. Contains events we are looking for
3: **for all** transitions t influenced by Δ_t with trigger $\sigma \in \Sigma_I$ **do**
4: $T_{visited} \leftarrow t$ ▷ Remember all visited transitions
5: $\Sigma_{current} \leftarrow \Sigma_{B_t}$ ▷ Set of currently active broadcast signals
6: $\Sigma_{influenced} \leftarrow \Sigma_{influenced} \cup$ FindIndirectInf(t, $\Sigma_{current}$, $T_{visited}$) ▷ Recursive analysis
7: **end for**
8: **return** $\Sigma_{influenced}$ ▷ Return the list of influenced events
9: **end procedure**

10: **procedure** FindIndirectInf(t,$\Sigma_{current}$,$T_{visited}$)
11: $\Sigma_{result} \leftarrow \emptyset$
12: $s_{current} \leftarrow$ target state of t
13: $\Sigma_{next} \leftarrow \emptyset$ ▷ Contains next broadcast events
14: $T_{out} \leftarrow \{t' \in T \mid \forall\, s \in S, t' = (s_{current}, s) \wedge t' \notin T_{visited} \wedge \sigma_{i_{t'}} \in \Sigma_{current}\}$▷ Find next transitions, which are not visited and are triggered by current events
15: $T_{visited} \leftarrow T_{visited} \cup T_{out}$ ▷ Mark transitions as visited
16: **for all** $t \in T_{out}$ **do**
17: $\Sigma_{next} \leftarrow \Sigma_{B_t}$ ▷ Update the current broadcast signals
18: **for all** $\sigma_B \in \Sigma_{B_t}$ **do**
19: **if** $\sigma_B \in \Sigma_O$ **then** ▷ Stop: Event corresponds to outgoing signal
20: $\Sigma_{result} \leftarrow \Sigma_{result} \cup \{\sigma_B\}$ ▷ Add signal to return set
21: $T_{out} = T_{out} \setminus \{t\}$ ▷ Remove the transition from the analyzed
22: **end if**
23: **end for**
24: **end for**
25: **if** $T_{out} = \emptyset$ **then** ▷ No more unvisited and valid successor transitions
26: **return** Σ_{result}
27: **else** ▷ Recursively visit other outgoing transitions
28: **return** $\Sigma_{result} \cup$ FindIndirectInf(t, Σ_{next},$T_{visited}$)
29: **end if**
30: **end procedure**

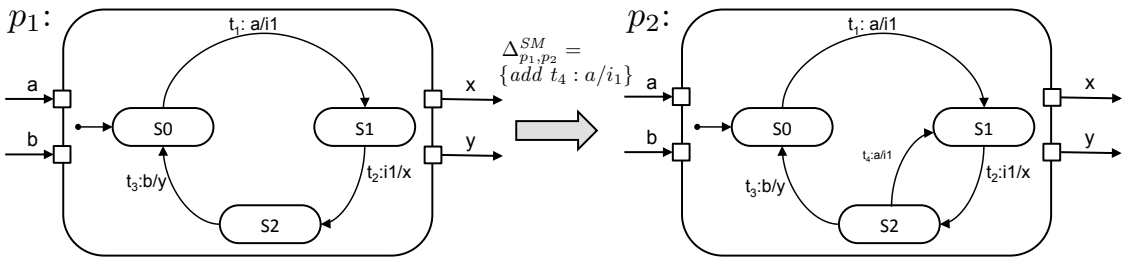

Figure 4.4: Example for Internal Component Changes

Definition 4.9: Indirect Signal Weight Metric

Let Π be the set of signals, $impact : \Pi \times \mathcal{P}(CON \times CON) \to \mathbb{N}$ the function which counts event occurrences for signals and $influence_c \subseteq I_c \times O_c$ the set of signal pairs where an incoming signal influences an outgoing signal in a component $c \in C$.

We define an *indirect signal weight metric* $m_{indirect_\pi} : C \times \Pi \to \mathbb{R}$ for a component $c \in C$ and a signal $\pi \in \Pi$ as follows:

$$m_{indirect_\pi}(c, \pi) = \begin{cases} \frac{impact(\pi, influence_c)}{|influence_c|} & if\ influence_c \neq \emptyset \\ 0 & if\ influence_c = \emptyset \end{cases}$$

Example 4.4: Computing Behavioral Weight Metrics

To illustrate the computation of behavioral weight metrics, Figure 4.4 shows the internal change of a component c between two variants p_1 and p_2. The delta $\Delta^{SM}_{p_1,p_2}$ only comprises one change operation, i.e., the addition of a transition t_4 as shown in Figure 4.4. Using the sample state machines, we give an example for the behavioral weight metric computation. First of all, notice that the interface of the component does not change, i.e., the architecture-based weight metrics are not able to detect a change between p_1 and p_2. However, as we show in the following, analyzing the changes of the internal behavior of the component leads to aspects important for testing.

Behavioral Component Weight. The behavioral component weight metric $m_{state}(c_1)$ is 0, as no state changes occurred between the two variants. However, there is one changed transition, i.e., we compute for the transition-based weight metric that $m_{tran}(c_1) = \frac{1}{4} = 0.25$.

Direct Signal Weights. For the signal-based weights, we assume that the analyzed signals only occur in the shown component. As transition t_4 is added in p_2, the corresponding delta directly influences the signals used by t_4, i.e., the incoming signal b and the outgoing signal x. Event i_1 is an internal event, which does not correspond to one of the four interface signals, i.e., we do not compute a signal weight for it. Accordingly, we compute a direct signal weight for signal a as $m_{direct_\pi}(a, p_2) = \frac{1}{4} = 0.25$.

Indirect Signal Weights. In the sample state machines shown in Figure 4.4, we detect an indirect signal influence. While the delta directly influences transition t_4, the change also has indirect influences. For our analysis, we initiate a path with t_4 as start. Firing t_4, we store i_1 as current event in $\Sigma_{current}$ and the current state $s_{current}$ is now $S1$. Here, one outgoing transition t_2 exists. Looking into $E_{current}$, we are able to fire this transition using the stored internal event i_1. This leads to the broadcast of event x, i.e., there exists an indirect influence relation between signals a and x. Here, the analysis stops, as an external event is send and no other transitions can be fired from $S1$. We compute the indirect signal weight for x as $m_{indirect_\pi}(c_1, x) = \frac{1}{1} = 1$.

4.4 Evaluation

We perform an evaluation of our testing framework using the BCS case study (cf. Chapter 3.2) to assess if it fulfills the requirements we specified. We use integration testing as evaluation scenario, i.e., we prioritize a set of integration test cases $\mathcal{TC}_{int} \subseteq \mathcal{TC}$ defined as MSCs. To this end, we first formulate research questions to guide our evaluation. Next, we explain the evaluation setup. Afterwards, the methodology of the evaluation is described followed by the results. Finally, we describe and debilitate potential threats to validity.

4.4.1 Research Questions

To assess the quality of our test case prioritization framework, we define the following three research questions to be evaluated.

RQ1: *How* feasible *is the test case prioritization framework in terms of its applicability for different testing artifacts?* We require the framework to be able to prioritize test cases for product variants of SPLs based on a set of defined weight metrics, weight functions and test case prioritization functions. Thus,

the framework shall be able to use different types of input data to produce priority values for test cases. In particular, we use integration testing as evaluation scenario.

RQ2: *How* extensible *is the framework in terms of supporting new artifacts for test case prioritization?* We require the framework to be extensible, i.e., new testing artifacts shall be integrable to provide additional test case prioritization approaches. To this end, we assess if additional testing data can be integrated. In particular, we integrate black-box meta-data, e.g., the failure-finding history of a test case, using the framework. In addition, we investigate if this integration mitigates the need to write new test case prioritization concepts and tools.

RQ3: *How* effective *are the resulting test case prioritization instances?* Using the framework, the user shall be able to prioritize test cases. Besides the availability of the prioritization functionality, we also require the results of the test case prioritization to be effective, i.e., to detect changed parts of the system as fast as possible.

4.4.2 Methodology

Setup for RQ1. To examine the quality of our test case prioritization framework and answer the first research question, we construct two different framework instances based on the delta-oriented and model-based weight metrics presented in Chapter 4.3 [LLS+17]. The main difference are the available input artifacts. The first instance only focuses on delta-oriented architectural test models and message sequence charts, which describe the test cases. Thus, it is suitable to describe if the framework is feasible for integration testing on architectural level. The second instance focuses on the extension of the test case prioritization to also consider delta-oriented state machines, which describe the internal behavior of components of the system. We argue that the framework has to be able to cope with additional artifacts and that sophisticated weight functions and test case prioritization functions have to be derivable.

Using these two instances, we are able to investigate the feasibility of our test case prioritization framework, i.e., if it is applicable to a wide range of different testing artifacts in different combinations while showing its adaptability. To this end, we create a set of weight functions and test case prioritization functions based on the provided weight metrics defined in Chapter 4.3.

Setup for RQ2. To investigate if the framework is actually extensible, which is one of the non-functional requirements we defined earlier (cf. Chapter 4.1), we

integrate another type of data. To this end, we chose data that is neither delta-oriented nor specifically model-based. In particular, we add *black-box meta-data*, which is available in testing, i.e., how many failures a test case has found in past executions, how often a test case has been when the test case has been executed for the last time. This information is relevant in regression testing [FKAP09, ERL11] and, thus, we argue that it is also useful for SPL testing. We assess, if we are able to integrate the data to further enhance the test case prioritization.

Setup for RQ3. To answer RQ3, we investigate the quality of the produced framework instances, i.e., how effective the test case orderings are. Unfortunately, failure information is not available for BCS. Thus, we have to use a different metric than *average percentage of faults detected* (APFD) (cf. Chapter 2.2.2) [RUCH01]. Therefore, we propose a new metric: *Average percentage of changes covered* (APCC) [LLL+15]. This metric is similar to APFD, but it measures the detection rate of changes between product variants rather than the failure finding rate. We measure system changes based on interfaces of a changed components, i.e., each connector connected to a changed component has to be retested as early as possible. This does not include connectors which are new, as we only focus on retest-scenarios of already tested parts of the system. In particular, the metric measures for a ordered test set $TS \subseteq \mathcal{TC}_{int}$ how fast a set of changes T_{change} is covered by the test cases in TS. For integration testing, changes comprise the set of connectors CON_{change}, which are connected to a component which has been modified by a first applied delta. This excludes connectors, which are added for the first time in the tested product variants, i.e., which are new, as new behavior has to be tested for product variant regardless of any prioritization. The values of APCC range between 0 and 1, where 1 corresponds to the best achievable result.

Definition 4.10: Average Percentage of Changes Covered (APCC)

Let $TS \subseteq \mathcal{TC}_{int}$ be a set of integration test cases, $CON_{change} = \{con_1, \ldots, con_m\}$ the set of changed connectors and T_{change_i} the i-th position where a changed connector is covered.

We define the *Average Percentage of Changes Covered* metric [LLL+15] for n test cases and m changed connectors as:

$$APCC = 1 - \frac{\sum_{i=1}^{m} T_{change_i}}{n \cdot m} + \frac{1}{2n}$$

We assess APCC for the instances shown in Table 4.1 using artifacts provided by BCS (cf. Chapter 3.2). We prioritize test cases, which are retestable for the current PUT, i.e., we analyze if they correspond to changes in the system and only prioritize relevant test cases. APCC is computed for each product variant separately.

Table 4.1: Overview of Framework Instances for RQ1

	1st Framework Instance	**2nd Framework Instance**
Available Artifacts	Delta-Oriented Architectural Test Models Message Sequence Charts	Delta-Oriented Architectural and Behavioral Test Models Message Sequence Charts
Component Weight Metrics	Incoming Connector Changes Outgoing Connector Changes MPD	Incoming Connector Changes Outgoing Connector Changes MPD Behavioral Component Weight
Signal Weight Metrics		Direct Signal Weights Indirect Signal Weights
Test Case Prioritization Functions	Component-based Communication-based	Component-based Communication-based Signal-based Dissimilarity-based

4.4.3 Results and Discussion

We investigate the quality of our testing framework by using BCS as case study. In the following, we present and discuss the results of our three research questions.

> **RQ1:** *How* feasible *is the test case prioritization framework in terms of its applicability for different testing artifacts?*

To assess the framework's feasibility, we construct two instances based on the provided weight metrics (cf. Chapter 4.3). We give an overview of the commonalities and differences of these two instances in Table 4.1. We have chosen these particular two instances, as they resemble previously custom-tailored solutions [LLL+15, LLAH+16]. To answer the first research question, we assess both instances in more detail in the following.

Assessing the First Framework Instance. The first framework instance (cf. second row in Table 4.1) uses delta-oriented architecture test models as specification for integration testing and MSCs as test cases for integration testing. Using our testing framework and the weight metrics for these artifacts, we are able to construct a *component weight function* $w : C \to \mathbb{R}$, which can be applied to compute different component weight for each component in the current PUT (cf. Definition 4.11).

Definition 4.11: Component Weight Function

Let $m_{in} : C \to \mathbb{R}$ be the weight metric for incoming connector changes, $m_{out} : C \to \mathbb{R}$ the weight metric for outgoing connector changes, $m_{MPD} : C \to \mathbb{R}$ the weight metric for MPDs and α, β and γ weighting factors.

We define the *component weight function* $w : C \to \mathbb{R}$ for component $c \in C$ as:

$$w(c) = \alpha \cdot m_{in(c)} + \beta \cdot m_{out}(c) + \gamma \cdot m_{MPD}(c),$$

such that $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \in \mathbb{R}$ holds.

The component weight function makes use of three weighting factors, α, β and γ. Thus, the impact of weight metrics can be defined by the user, which allows for more *controllability* of the framework even after an instance has been implemented.

Using the component weight function, we derive two different test case prioritization functions for the first framework instance [LLL+15, LLS+17]. Both are applied to test cases which are reusable for the currently tested product variant p. The set of all reusable integration test cases for a product variant is defined as $\mathcal{TC}_{reuse_p} \subseteq \mathcal{TC}_{int}$.

First, we design a *component-based test case prioritization function* $prio_{comp} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$. The intention of this function is to give a higher priority to MSCs, which test the interaction of highly weighted components as these components have been changed more drastically, which can prevent their correct communication. The components covered by a test case tc are referred to as $C_{tc} \subseteq C$. Basically, it sums the weight values of all covered components up to assess the priority of a test case. This is normalized by the number of covered components C_{tc} to avoid that large test cases automatically receive a higher weight.

Definition 4.12: Component-Based Test Case Prioritization Function

Let $\mathcal{TC}_{reuse_p} \subseteq \mathcal{TC}_{int}$ be the set of integration test cases reusable in product variant $p \in P_{SPL}$, $c \in C_{tc}$ a component covered by test case $tc \in \mathcal{TC}_{int}$ and $w : C \to \mathbb{R}$ the component weight function for a component $c \in C$.

We define the *component-based test case prioritization function* $prio_{comp} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$ which computes the priority of a test case $tc \in \mathcal{TC}_{reuse_p}$, applicable to product variant $p \in P_{SPL}$ as:

$$prio_{comp}(tc, p) = \left\{ \frac{\sum_{j=1}^{|C_{tc}|} w(c_j)}{|C_{tc}|} \;\middle|\; c_j \in C_{tc} \right\}$$

While the component-based test case prioritization function is able to prioritize test cases, it is a trivial function as it does not consider the actual scenario described by an MSC. Thus, we define a *communication-based test case prioritization function* $prio_{comm} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$, which takes the signals exchanged between components in an MSC into account. The computation is also based on the component weight metric. However, the test case prioritization function assesses which components communicate with each other and how often, using their exchanged signals. In particular, it assesses the multi-set of covered tested signal exchanged, which are returned by the function $s : C \times C \to \mathcal{P}(\Pi)$ for a test case. We require the result to be multi-set to compute the exact number of interactions of components.

Definition 4.13: Communication-Based Test Case Prioritization Function

Let $\mathcal{TC}_{reuse_p} \subseteq \mathcal{TC}_{int}$ be the set of integration test cases reusable in product variant $p \in P_{SPL}$, n the number of all signals $\pi \in \Pi$ covered by a test case $tc \in \mathcal{TC}_{int}$, $s : C \times C \to \mathcal{P}(\Pi)$ the function to return the multi-set of signals between two components covered in the test case and $w : C \to \mathbb{R}$ the component weight function for a component $c \in C$.

We define the *communication-based test case prioritization function* $prio_{comm} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$ to derive the priority of a test case $tc \in TC_{reuse_p}$ in product variant $p \in P_{SPL}$ as:

$$prio_{comm}(tc, p) = \frac{\sum_{j=1}^{n} \sum_{m=1}^{n} |s(c_j, c_m)| \cdot (w(c_j) + w(c_m))}{\sum_{j=1}^{n} \sum_{m=1}^{n} |s(c_j, c_m)|},$$

such that $c_j, c_m \in C_{tc}$ holds.

Assessing the Second Framework Instance. In addition to the architectural artifacts, we investigate the capability of our testing framework to deal with behavioral artifacts. In particular, we integrate delta-oriented state machines as further artifacts to be used by the test case prioritization [LLAH+16]. Using our testing framework, we can apply existing definitions provided earlier. This allows for an efficient definition of new weight and test case prioritization functions, which omits the necessity to create a new test case prioritization implementation from scratch.

We first implement a *behavioral component weight function* $w_b : C \to \mathbb{R}$. It assesses the internal changes of a component c based on the provided state machine deltas. We assess the weight of a component based on the number changes of its states and transitions based on the two behavioral weight metrics defined in Chapter 4.3. Our testing framework allows us to integrate this new behavioral component weight function easily into the existing weight function, using weighting factors.

Definition 4.14: Behavioral Component Weight Function

Let $m_{in} : C \to \mathbb{R}$ be the weight metric for incoming connector changes, $m_{out} : C \to \mathbb{R}$ the weight metric for outgoing connector changes, $m_{MPD} : C \to \mathbb{R}$ the weight metric for MPDs, $m_{state} : C \to \mathbb{R}$ the state change weight metric, $m_{tran} : C \to \mathbb{R}$ the transition change weight metric, $w : C \to \mathbb{R}$ the component weight function and $\alpha, \beta, \gamma, \theta$ and ζ weighting factors.

We define the *behavioral component weight function* $w_b : C \to \mathbb{R}$ for a component $c \in C$ as follows:

$$w_b(c) = \alpha \cdot m_{in(c)} + \beta \cdot m_{out}(c) + \gamma \cdot m_{MPD}(c) + \theta \cdot m_{state}(c) + \zeta \cdot m_{tran}(c),$$

such that $\alpha + \beta + \gamma + \theta + \zeta = 1$ and $\alpha, \beta, \gamma, \theta, \zeta \in \mathbb{R}$ holds.

While the new behavioral component weight function is a simple addition to the previously defined component weight function, it provides a lot of flexibility. The user can define which weight metrics are of importance, e.g., if only internal changes occur for the PUT, then the weight factors α, β and γ should be set to 0. This supports the controllability of the framework's instances and the outcome of the test case prioritization, without the need to reimplement anything.

In addition, we define a signal weight function, which gives a weight value to a signal $\pi \in \Pi$ of the current system. To this end, we combine the direct and indirect signal weight metrics defined in Chapter 4.3.2. The construction of the resulting *signal weight function* $w_{sig} : C \times \Pi \to \mathbb{R}$ is similar to the previously described weight functions. It can be used to weight the signals used in an MSC separately based on the behavioral changes of the tested components.

Definition 4.15: Signal Weight Function

Let Π_c be the set of signals used in component $c \in C$, $m_{direct_\pi} C \times \Pi \to \mathbb{R}$ the direct signal weight metric, $m_{indirect_\pi} C \times \Pi \to \mathbb{R}$ the indirect signal weight metric and μ, λ weighting factors.

We define the *signal weight function* $w_{sig} : C \times \Pi \to \mathbb{R}$ for a component $c \in C$ and a signal $\pi \in \Pi_c$ as follows:

$$w_{sig}(c, \pi) = \lambda \cdot m_{direct_\pi}(c, \pi) + \mu \cdot m_{indirect_\pi}(c, \pi),$$

such that $\lambda + \mu = 1$ and $\lambda, \mu \in \mathbb{R}$ holds.

Based on our existing communication-based test case prioritization function, we define a signal-based test case prioritization function $prio_{sig} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$, which prioritizes MSCs based on their exchanged signals, the weights of the involved components and the direct and indirect signal weights. Priorities are normalized over the set of covered signals to avoid that large test cases always receive a high priority.

Definition 4.16: Signal-Based Test Case Prioritization Function

Let $\mathcal{TC}_{reusable_p} \subseteq \mathcal{TC}_{int}$ be the set of test cases reusable for product variant $p \in P_{SPL}$, Π_{tc} the number of multi-set of signals covered by a test case tc, $s : C \times C \to \mathcal{P}(\Pi)$ the function to return the multi-set of signals between two components covered in a test case, $w : C \to \mathbb{R}$ the component weight function and $w_{sig} : C \times \Pi \to \mathbb{R}$ the signal weight function.

We define a *signal-based test case prioritization function* $prio_{sig} : \mathcal{TC}_{int} \times P_{SPL} \to \mathbb{R}$ for a test case $tc \in \mathcal{TC}_{int}$ in product variant $p \in P_{SPL}$ as:

$$prio_{sig}(tc, p) = \frac{\sum_{m=1}^{|\Pi_{tc}|} \sum_{j=m+1}^{|\Pi_{tc}|} (w(c_m) + w(c_j) + \sum \{k \cdot (w_{sig}(c_m, \pi) \cdot w_{sig}(c_j, \pi)) \mid \Pi_{mj}(\pi) = k\})}{\sum_{m=1}^{n} \sum_{j=1}^{n} |\Pi_{mj}|}$$

with $\Pi_{mj} = s(c_m, c_j)$.

We define an additional test case prioritization function, which is based on the *dissimilarity* of test cases. Dissimilarity-based testing aims to focus on test cases which differ the most from each other and, thus, have a higher likelihood to reveal new failures [AHTM+14, NH15]. We define a *dissimilarity function* $dissim : \mathcal{TC} \times \mathcal{TC}_{int} \to \mathbb{R}$, which compares two MSCs in terms of their similarity based on the Jaccard-Distance. In this thesis, two MSCs are similar if they exchange the same signals between the same components.

Definition 4.17: Test Case Dissimilarity Function

Let $\mathcal{TC}_{int}$ be the set of integration test cases, $i, j \in \{1, \ldots, |\mathcal{TC}_{int}|\} \in \mathbb{N}$ two indices and Π_{tc} the multi-set of signals covered by a test case.

We define a *test case dissimilarity function* $dissim : \mathcal{TC}_{int} \times \mathcal{TC}_{int} \to \mathbb{R}$ for two test cases $tc_i, tc_j \in \mathcal{TC}_{int}$ as follows:

$$dissim(tc_i, tc_j) = 1 - \frac{|\Pi_{tc_i} \bigcap \Pi_{tc_j}|}{|\Pi_{tc_i} \bigcup \Pi_{tc_j}|}$$

When prioritizing a test set, we aim to select the test case which is most dissimilar to all previously prioritized test cases, i.e., test cases which already have been added to the test set according to their dissimilarity. The idea behind this is to cover the overall system as fast as possible, as each dissimilar test case covers different parts of the system. To this end, we design a *dissimilarity-based* test case prioritization function $prio_{dissim} : \mathcal{TC}_{int} \times \mathcal{P}(\mathcal{TC}_{int}) \to \mathbb{R}$. It prioritizes test cases according to their dissimilarity to the set of already prioritized test cases. In other words, a test case tc is compared to all previously prioritized test cases TC_{prio}. This is repeated for all unprioritized test cases. The most dissimilar test case is selected next. Definition 4.18 defined the prioritization function. Of course, only test cases, which are applicable to current PUT should be compared and prioritized.

Definition 4.18: Dissimilarity-Based Test Case Prioritization Function

Let $\mathcal{TC}_{int}$ be the set of integration test cases, $dissim : \mathcal{TC}_{int} \times \mathcal{TC}_{int} \to \mathbb{R}$ be the test case dissimilarity function and $TC_{prio} \subseteq \mathcal{TC}$ the set of already prioritized test cases.

We define the *dissimilarity-based test case prioritization function* $prio_{dissim} : \mathcal{TC}_{int} \times \mathcal{P}(\mathcal{TC}_{int}) \to \mathbb{R}$ for a test case $tc \in \mathcal{TC}_{int}$ and a set of already prioritized test cases TC_{prio}, such that $tc \notin TC_{prio}$ holds, as follows:

$$prio_{dissim}(tc, TC_{prio}) = \frac{\sum_{n=1}^{|TC_{prio}|} dissim(tc, tc_n)}{|TC_{prio}|}$$

The dissimilarity-based test case prioritization supports the other test case prioritization functions defined earlier in a particularly useful way, as it reduces the chance of a ”clustering” of test cases. Clustering occurs, if test cases are very similar and, thus, receive similar priority values reducing the coverage rate of different system parts. Using the dissimilarity-based approach in combination with delta-oriented test case prioritization functions leads to a combination of both, fast coverage of different parts of the system as well as a focus on important changes in the system.

Based on the provided weight and test case prioritization functions, we are able to validate the first research question, showing that our test case prioritization framework for SPLs is indeed applicable to different model-based testing artifacts based on the same input data without a need for creating new test case prioritization implementations from scratch. Instead, it supports the extension of existing definitions and the reuse of test artifacts.

RQ2: *How* extensible *is the framework in terms of supporting new artifacts for test case prioritization?*

The second research question aims to assess the extensibility of our testing framework in terms of supporting additional artifacts. In contrast to the first research question, we do now focus on test artifacts which are not SPL specific. In particular, we investigate if *black-box meta-data*, as available in system testing, can be incorporated to support the test case prioritization. While this data is typical in testing, it is only one example of how the test case prioritization can potentially be extended.

To this end, we first describe the artifacts which we assume are available during system testing [Sne07] as described in Chapter 3.1.2, i.e., the test case age, last test case execution and its previous failure finding history. We present the usage of available data for SPL integration testing scenario in the following.

Test Case Age. We argue, that a test case which has been applied to a high number of product variants has a low likelihood to reveal a new failure. Thus, we count the number of occurrences for a test case for previously tested variants in our incremental approach. Of course, the analyzed test case tc has to be in the set of reusable test cases for the current PUT p, i.e., $tc \in \mathcal{TC}_{reuse_p}$. The *test case age* does not include the actual number of executions, but the numbers of times the test case has been applicable to previously test product variants. As this particular information is not mappable to elements of the architecture, we do not define weight metrics or functions, but directly specify a *test case age prioritization function* $prio_{age} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$, where the number of occurrences of the tc performs as weight metric. If the ratio of applicability to number of previous product variants is very high for a test case, its priority is high. We normalize the value to between 0 and 1, where 1 indicates the highest priority value.

Definition 4.19: Test Case Age Prioritization Function

Let $TC_{reuse_p} \subseteq \mathcal{TC}_{int}$ the set of test cases reusable for a product variant $p \in P_{SPL}$.

We define the *test case age prioritization function* $prio_{age} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ for an integration test case $tc \in \mathcal{TC}_{int}$ based on the set of previously tested product variants $P_{tested} \subseteq P_{SPL}$ as follows:

$$prio_{age}(tc, P_{tested}) = \frac{1}{1 + \sum_{j=1}^{|P_{tested}|} \begin{cases} 1, & \textit{iff } tc \in TC_{reuse_{p_j}} \\ 0, & \textit{otherwise} \end{cases}}$$

Last Test Case Execution. We assume that a test case, which has not been executed for a long time, has a higher likelihood to reveal a new failure than a recently executed test case. This ensures that functionality which has not been tested for a long time is tested again. Due to the complexity of SPLs and the potentially large number of product variants under test, we do not measure the last execution in *time*, but in the number of variants under test since the test case has been executed last. We define a function $exec : \mathcal{TC}_{int} \times P_{SPL} \to \{0, 1\}$ which returns 1 if a test case has been executed for a given product variant, and 0 otherwise. The more product variants have been tested without executing a particular test case tc, the higher shall the likelihood be to test it again. Consequently, we define a test case prioritization function $prio_{lastexec} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$, which counts the number of variants since the last execution of a test case. If the test case is executed for a product variant, the value is set to 0.

Definition 4.20: Last Test Case Execution Prioritization Function

Let $\mathcal{TC}_{int}$ be the set of test cases and P_{SPL} the set of product variants.

We define $exec : \mathcal{TC}_{int} \times P_{SPL} \to \{0, 1\}$ to be the execution function for a test case $tc \in \mathcal{TC}_{int}$ in product variant $p \in P_{SPL}$ as follows:

$$exec(tc, p) = \begin{cases} 1 & \textit{if } tc \textit{ was executed in } p \\ 0 & \textit{otherwise} \end{cases}$$

We define the *last test case execution prioritization function* $prio_{lastexec} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$ to prioritize an integration test case $tc \in \mathcal{TC}_{int}$, the set of previously tested product variants $P_{tested} \subseteq P_{SPL}$ and according to their last execution as follows:

$$prio_{lastexec}(tc, P_{tested}) = |\{j \mid i = max(k \mid exec(tc, p_k) = 1) \wedge i < j \leq |P_{tested}|\}|$$

Failure Finding History. The *effectiveness* of a test case is interesting for prioritization, i.e., how many failures it has revealed in the past. Test cases which previously revealed failures are important for regression testing, e.g., to ensure that a bug has been fixed [ERL11]. Similarly, in SPLs a failure found in a previous product variant might lead to a fix that influences other product variants as well. We define the function $revealed : \mathcal{TC}_{int} \times P_{SPL} \to \mathcal{P}(\mathcal{F})$, which returns the set of failures which have been revealed by a test case $tc \in \mathcal{TC}_{int}$ in product variant $p \in P_{SPL}$. Therefore, we introduce a test case prioritization function $prio_{history} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$, which assesses the quality of a test case based on the number of failures it revealed in past executions.

Definition 4.21: History-Based Test Case Prioritization Function

Let $\mathcal{F}$ be the set of revealed failures, $exec : \mathcal{TC}_{int} \times P_{SPL} \rightarrow \mathcal{P}(\mathcal{TC}_{int})$ the function, which returns if a test case $tc \in \mathcal{TC}_{int}$ has been executed in product variant $p \in P_{SPL}$ and $revealed : \mathcal{TC}_{int} \times P_{SPL} \rightarrow \mathcal{P}(\mathcal{F})$ the function, which returns the set of revealed failures for a certain product variant by a test case tc.

We define the *history-based test case prioritization function* $prio_{history} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ to prioritize a test case $tc \in \mathcal{TC}_{int}$ based on the set of tested product variants $P_{tested} \subseteq P_{SPL}$ according to previously revealed failures as:

$$prio_{history}(tc, P_{tested}) = \begin{cases} \frac{\sum_{i=1}^{|P_{tested}|} |revealed(tc,p_i)|}{\sum_{j=1}^{|P_{tested}|} exec(tc,p_j)} & if\ \exists p_j \in P_{tested} : exec(tc,p_j) > 0 \\ 0 & otherwise \end{cases}$$

Integrating Meta-Data into Framework. Our test case prioritization framework for SPLs allows us to combine test case prioritization functions similar to the combination of weight metrics by adding weighting factors. Using the earlier presented prioritization functions, we are able to create a test case prioritization function $prio_{meta} : \mathcal{TC}_{int} \times P_{SPL} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$, which incorporates behavioral and meta-data information in combination. It computes the priority of a test case tc based on the previously tested product variants P_{tested}.

Definition 4.22: Meta-Data Test Case Prioritization Function

Let $prio_{sig} : \mathcal{TC}_{int} \times P_{SPL} \rightarrow \mathbb{R}$ be the signal-based test case prioritization function, $prio_{age} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ the test case age prioritization function, $prio_{lastexec} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ the last test case execution prioritization function, $prio_{history} : \mathcal{TC}_{int} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ the history-based test case prioritization function and $\alpha, \beta, \gamma, \theta \in \mathbb{R}$ weighting factors.

We define the *meta-data test case prioritization function* $prio_{meta} : \mathcal{TC}_{int} \times P_{SPL} \times \mathcal{P}(P_{SPL}) \rightarrow \mathbb{R}$ for an integration test case $tc \in \mathcal{TC}_{int}$ in product variant p and the set of preciously tested product variant $P_{tested} \subseteq P_{SPL}$ as follows:

$$prio_{meta}(tc, p, P) = \\ \alpha \cdot prio_{sig}(tc, p) + \beta \cdot prio_{age}(tc, P) + \gamma \cdot prio_{lastexec}(tc, P) + \theta \cdot prio_{history}(tc, P),$$

such that $P = P_{tested}, p \notin P$ and $\alpha + \beta + \gamma + \theta = 1$ holds.

Consequently, we validate our second research question, showing that additional testing data can be integrated easily. Even though the sample data was not model-based nor delta-oriented, we were able to create a suitable test case prioritization function which also integrates the existing framework artifacts.

RQ3: *Are the the resulting test case prioritization instances* effective*?*

The third and last research question assesses the quality of the resulting test case prioritization functions derived by our testing framework for SPLs. In other words, we examine how effective the produced orderings of test cases are. To this end, we measure the APCC metric (cf. Chapter 4.4.2) for the 18 product variants provided by the BCS case study. One exception is the core variant $P0$, for which no test cases are prioritized, as we only prioritize reusable test cases. However, for the core every test case is new and, thus, has to be executed regardless of a test case priority.

We evaluate different framework instances, based on the architecture-based, signal-based, dissimilarity-based and meta-data-based test case prioritization functions and particular combinations. In total, we analyze nine different framework instances. We selected four representative instances which is explained in more detail in the following. In particular, we explain the *Communication-based* (*ComB*), *dissimilarity-based* (*DB*), *Component-based and Meta-Data-based* (*CB and MD*) and the *Communication-based and dissimilarity-based* (*ComB and DB*) test case prioritizations. Using these instances, we are able to show the effects of the dissimilarity-based idea in combination with delta-oriented approaches, as well as the extension of meta-data.

We present the average APCC results of the four different framework instances and the random technique in Figure 4.5. The figure shows a bar chart, where the x-coordinate represents the BCS product variants $P1$ to $P16$. $P17$ is missing, as no new elements have been found, making a prioritization unnecessary. The y-coordinate represents the achieved APCC value for five different techniques for the product variants. For our evaluation, we applied different test case prioritizations based on different weight metrics in addition to a random approach. For each technique, we measure the APCC values for each product variant under test, excluding the core variant for which no prioritization is performed. In addition, we present the average APCC values over the set of all 16 product variants produced by the five different techniques in Table 4.2. The detailed results of all analyzed framework instances are shown in Appendix A in separate figures.

Looking at the results presented in Figure 4.5, we can see that different test case prioritization instances produce a different outcome in terms of quality. First of all, we establish a baseline in form a random testing approach. We perform 100 random orderings of applicable test cases for each product variant and normalize the APCC

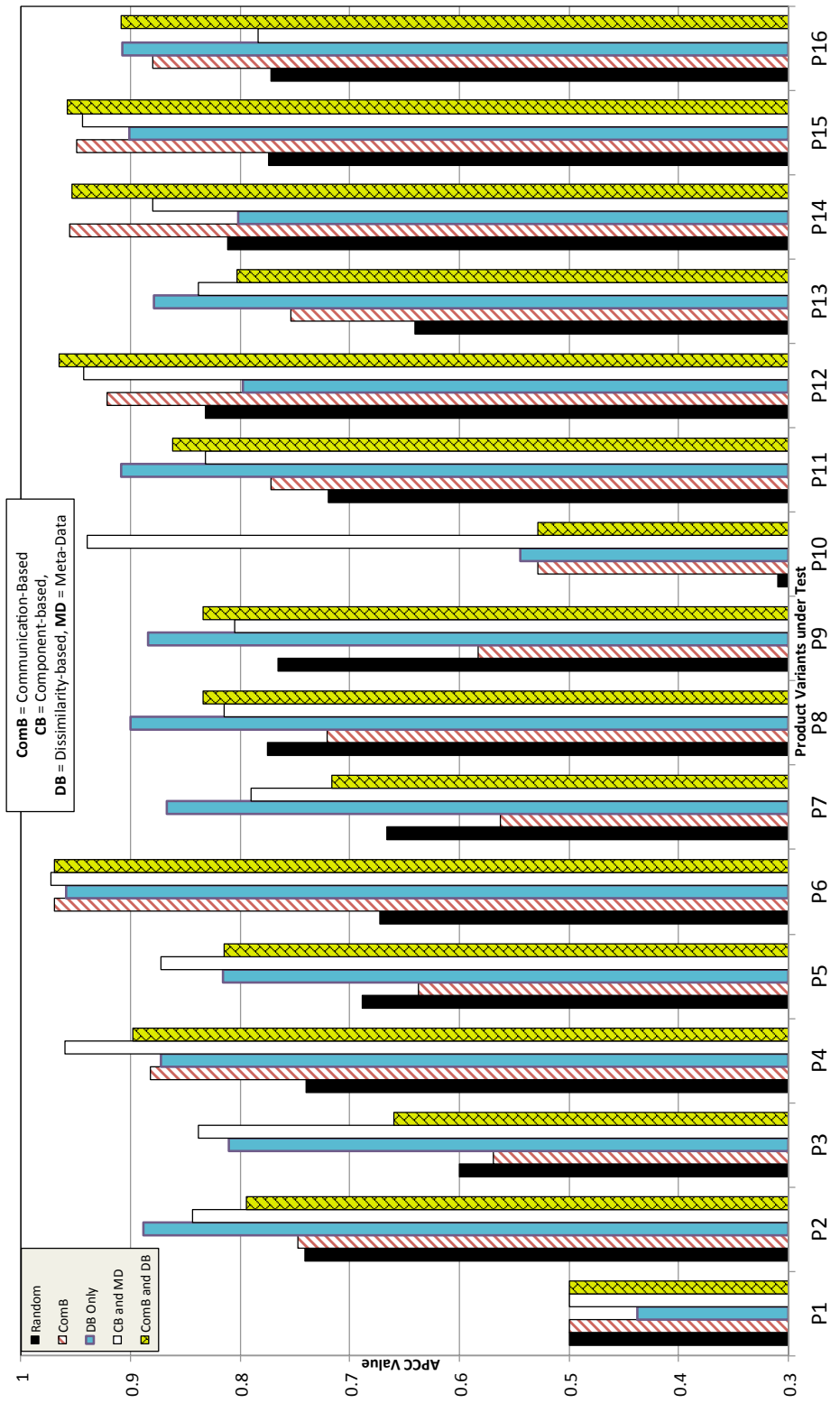

Figure 4.5: APCC Results for Framework Instances (cf. [LLS+17])

Table 4.2: Overview of Average APCC Results for BCS

Random	*ComB*	*DB Only*	*CB and MD*	*ComB and DB*
0.688	0.723	0.823	0.847	0.812

ComB = Communication-Based, **CB** = Component-Based, **DB** = Dissimilary-Based, **MD** = Meta-Data

Table 4.3: Configuration of Evaluated Instances

Parameter	**CB / ComB**	**CB and MD**
Incoming Connector Changes	0.5	0.25
Outgoing Connector Changes	0.25	0.15
MPD	0.25	0.1
Test Case History	0	0.2 (N.A.)
Test Case Execution	0	0.2
Test Case Age	0	0.1

results. In average, the random test case prioritization achieves an APCC of 0.688, which is the lowest score (cf. Table 4.2). However, in certain cases the random technique is able to outperform other techniques, e.g., the component-based test case prioritization (cf. Appendix A for detailed results). This is most likely due to the clustering effect, i.e., that similar test cases receive a high prioritization value, which reduces the chance to detect all changes early.

The configurations of weighting factors of our architecture-based framework instances are shown in Table 4.3. We argue, that incoming connector changes are more important than outgoing changes, which is why we double the impact value of the corresponding weight metric compared to the outgoing connector changes (i.e., 0.5 vs. 0.25). MPDs are of minor importance to us, as they are a special case and, thus, only receive a weight of 0.25 as well. Dissimilarity-based testing is integrated by reducing the shown values by half and giving the dissimilarity-based prioritization the same importance as the shown prioritization functions in Table 4.3.

In Figure 4.5, we show results of five different techniques. The random technique is outperformed by the other techniques for most product variants. However, it outperforms the communication-based technique in some cases (e.g., $P7$). We deduce that these product variants have very similar test cases of high priority, leading to the earlier described clustering effect. We also perform a solely dissimilarity-based test case prioritization (*DB Only*), which achieves very good APCC values. However, this is the only approach which has to order all reusable test cases, instead of

being performed on the subset of retestable test cases. The latter can be only identified when using delta-oriented approaches. Test cases, which do not correspond to changes in the system do not have to be prioritized, which is the reason why *DB* is the only technique to achieve an APCC worse than 0.5 in the $P1$, where only one test case is retestable. The communication-based test case prioritization (*ComB*) is slightly better than the component-based test case prioritization in terms of average APCC, which shows that the analysis of the test cases is very useful. Based on RQ2, we use meta-data in combination with the component-based prioritization (*CB and MD*). This increases the performance compared to the other techniques. Our configuration for this instance is shown in Table 4.3. For this particular instance, no failure data was available for meta-data, i.e., we are only able to use the test case age and last execution weight metrics. As Figure 4.5 indicates, this technique achieves by far the best score for product variant $P10$. This is due the fact, that out of ten changed components six also have changed interfaces, which is the highest degree of changes in a product variant for BCS. Here, the component-based technique leads to the identification of important parts, while the meta-data ensures the coverage of solely tested parts. In average, the meta-data and component-based test case prioritization achieves the highest APCC value of all analyzed techniques with an average of 0.847 (cf. Table 4.2). The last technique, for which Figure 4.5 shows the results, is a combination of the communication-based and dissimilarity-based approach (*ComB and DB*). This technique outperforms the solely communication-based test case prioritization (*ComB*).

We analyzed further techniques, especially the signal-based techniques. However, for BCS, the signal-based test case prioritization did not produce effective orderings as their are no internal events in the state machines, prohibiting the analyzes of indirect weights. We present the detailed results of the other techniques in Appendix A.

Overall, the instances derived from our test case prioritization framework are able to produce useful and improved test case prioritizations in terms of fast coverage compared to a random approach. They do also scale with a rising number of product variants under test, as the performance usually increases while advancing through the set of product variants under tests, showing better results for later product variants. This is due to the fact, that the reuse potential increases with a larger number of product variants in P_{tested} as more features and parts of the system have already been tested.

Significance Test. We analyze the statistical significance of our evaluation results. To this end, we compute the *Mann-Whitney-U-Test* [MW47] for each pair of the instances we analyzed. We notice that, for a large number of test case prioritizations, the differences of their resulting orderings are not statistically significant, which has several reasons. For one, the data set is rather small, which reduces the number of different APCC results. Furthermore, we provide instances based on one

weight metric (e.g., component-based) and the combination with dissimilarity-based which leads to similar results. However, we observe combinations which are very different. Thus, the framework is able to produce different results, depending on the defined prioritization functions. The detailed p-values are provided in Table A.1 and in Table A.2 in Appendix A.

Summarizing, we validate our third research question, showing that the resulting APCC values are more effective than a random test case prioritization. In addition, the integration of additional data improves the test case prioritization in terms of effectiveness, i.e., the resulting APCC values.

4.4.4 Satisfaction of Requirements

In Chapter 4.1, we defined a set of functional and non-functional requirements for our test case prioritization framework to be fulfilled. Using the results of our evaluation, we assess whether these requirements and, thus, the framework's specification have been fulfilled.

Functional Requirements. We defined three functional requirements: *analysis of delta-oriented test models (FR1)*, *definition of weight metrics for test models elements (FR2)* and *flexible definition of test case prioritization functions (FR3)*. In RQ1 we examined the feasibility of our testing framework for architectural artifacts. We were able to show that the framework is applicable to delta-oriented architecture models and MSCs, which fulfills requirement *FR1*. In addition, we defined a set of weight metrics for these types of test models and test cases, which further fulfills *FR2*. We defined several test case prioritization functions for architectural and behavioral test models as well as meta-data, validating the third functional requirement *FR3*. Consequently, we argue that the intended functionality and three defined functional requirements for the framework are fulfilled.

Non-Functional Requirements. We defined three non-functional requirements for our testing framework: *extensibility (NFR1)*, *controllability (NFR2)* and *usability (NFR3)*. The second research question directly addresses the extensibility of the framework. The corresponding results show, that additional data, in form of black-box meta-data, can be easily integrated into our test case prioritization framework. This fulfills requirement *NFR1*. The second non-functional requirement *NFR2* states, that derived weight and test case prioritization functions shall be controllable by the user, i.e., even after framework instances have been derived, the test case prioritization has to be adaptable to the user's needs. This is enabled due to the introduced weighting factors, with which we can control the output of the test case prioritization, e.g., by disabling certain weight metrics. This ensures the required controllability and requirement *NFR2*. The third non-functional requirements *NFR3* demands *usability* of our testing framework, i.e., integrating new test

case prioritization approaches shall be more efficient than creating everything from scratch. Both the first and second research question have validated this, as they show that test case prioritization functions are easily defined and combinable. The third research question shows that the results are effective, i.e., the resulting test case prioritization outperform a random approach. We compared these results with results derived by earlier versions of the test case prioritization, which have been created from scratch [LLL+15, LSN+16], not finding any differences, i.e., we are able to produce high quality results similar to custom-tailored software, without the high effort associated with creating test case prioritization concepts and software from scratch.

4.4.5 Threats to Validity

We identified the following internal and external threats to validity for our evaluation and performed experiments [RH09].

Internal Threats. As our test case prioritization framework for SPLs requires different input data, the data quality heavily influences the results of our resulting test case prioritization. To mitigate negative aspects related to input data, we use existing test models which have been evaluated in previous techniques [LLLS12, LLL+14]. While we only use two different types of test models for evaluation, we show the applicability of our test case prioritization framework using state machines and architecture test models, which we assume to be available in model-based unit and integration testing, respectively. These two model types also represent structural and behavioral models, which are some of the most prominent model types in model-based testing [UL07]. Thus, we argue that our evaluation shows the applicability of our testing framework to common types of test models. As our test case prioritization framework for SPLs and corresponding prototype instances are developed manually, there can be problems within the software which could influence the gathered results. To tackle these issues, we tested our implementation throughly.

External Threats. While we carefully designed, executed and documented our experiments, we only had access to one case study for evaluation. This reduces the generalizability of gathered results. Further case studies have to be performed to support our claims. However, we are able to show positive results for the given case study, which offers a wide variety of artifacts, including different test models and black-box meta-data. Consequently, the evaluation lays the foundation for future evaluations, showing that the framework fulfills its specification.

4.5 Related Work

Regression testing is a common task in software engineering [Som10]. Yoo and Harman [YH07b] conducted a survey about the three major regression testing approaches, i.e., test case prioritization, test case selection and test case minimization, showing a wide range of existing techniques for single-software systems, which mostly focus on code analysis, e.g., by analyzing structural coverage [RUCH99]. These techniques are not focusing on SPL scenarios, i.e., they do not consider variability.

For SPL testing, several approaches have been proposed to improve testing. This has been shown by a survey on SPL testing techniques conducted by Engström and Runeson [ER11]. They conclude their research with a need for new SPL techniques which further exploit the characteristics of SPLs to improve SPL testing, especially when dealing with a large number of product variants. Machado et al. [MMCDA14] performed another survey on SPL testing strategies. The authors claim that there is still a need for techniques and that there is a lack of generalization of existing approaches. Many SPL testing techniques are applied on product level, i.e., they analyze feature configurations to provide product variant samples, which reduces the number of product variants under test [JHF12, LOGS11, OZLG11] or they prioritize product variants [AHTM+14, DPC+13]. However, these techniques do not provide information on how to test the retrieved product variants. Muccini and van der Hoek [MvdH03] analyze the potential of integration testing for SPLs, which is the scope of our testing framework for SPLs. They compare potential techniques to approaches in single-software systems. Concluding, they see a need for reuse of common artifacts to support SPL integration testing.

To show the novelty of our testing framework, we present related work in SPL testing, which also use regression testing approaches or artifacts similar to our testing framework. An overview of the presented related work is given in Table 4.4.

Product-based SPL Testing. First, we present SPL testing techniques which use similar artifacts and ideas as our test case prioritization framework, but focus on product level of SPLs. Thus, they do not provide information about testing of individual product variants but rather on the selection or prioritization of product variants. **Al-Hajjaji et al.** [AHTM+14] present a dissimilarity-based prioritization approach for product variants. They analyze the feature configurations of each product variant under test and select the product variant next, which is most dissimilar to all previously tested product variants. This is a similar concept to our dissimilarity-based test case prioritization function, avoiding clustering and redundancy in testing. However, they do only consider features as input and prioritize whole product variants instead of test cases for each product variant, without the focus on extensibility. **Al-Hajjaji et al.** [AHLL+17] extend their dissimilarity-based prioritization with the analysis of delta-oriented product variants. Each product

Table 4.4: Categorization of Related Work for SPL Testing Framework

Related Work	Technique			Input Artifacts					
	Select.	**Prio.**	**TC-Level**	**MBT**	**Deltas**	**MD**	**FC**	**Other**	**Ext.**
Product-based SPL Testing									
Al-Hajjaji et al. [AHTM+14]		X					X		
Al-Hajjaji et al. [AHLL+17]		X		X	X				
Devroey et al. [DPC+13]		X						X	
Ensan et al. [EBA+11]	X	X					X	X	
Lopez-Herrejon et al. [LHJFC+14]	X	X					X		
Muccini [Muc07]	X			X					
Parejo et al. [PSS+16]		X				X	X	X	(X)
Qu et al. [QCR08]	X	X					X	X	
Test Case Related SPL Testing									
Arrieta et al. [AWSE16]		X	X			X		X	
Baller et al. [BL14, BLLS14]	(X)	X	(X)			X		X	(X)
Knapp et al. [KRS15]	X		X						
Lackner [Lac15]	X		(X)				X	X	
Lity et al. [LMTS16]	X		X	X	X				
Lochau et al. [LLL+14]	X		X	X	X				
Neto et al. [DMSNdCMC+10]	X	X	X	X				X	
Reis et al. [RMP07]	(X)		X	X					
Uzuncaova et al. [UKB10]	(X)		X					X	
Varshosaz et al. [VBM15]	(X)		X	X	X			(X)	
Wang et al. [WAG13, WAG15]	(X)		X			X	X	X	
Our SPL Testing Framework	(X)	X	X	X	X	X		(X)	X

Select. = Selection techniques, **Prio.** = Prioritization techniques, **TC-Level** = Technique applicable to test cases,
MBT = Model-based testing approach, **MD** = Meta-data, **FC** = Feature configurations, **Ext.** = Technique is designed to be extensible

variant is represented as delta model. They show their technique using architecture test models, similar to our evaluation scenario. However, they do not restrict their techniques to architecture models. Results indicate that the differences based on deltas instead of feature configurations improve the failure finding rate compared to their previous approach. In contrast to our testing framework, they solely focus on dissimilarity analysis and prioritize product variants instead of test cases. **Devroey et al.** [DPC+13] use a dissimilarity-based search-based approach to prioritize product variants. Their concept is based on feature-transition systems, which they use to compute the dissimilarity of product variants. Compared to this thesis, they perform a prioritization of product variants instead of test cases, while also using model-based approaches. In addition, their approach is not designed to be extensible nor does it consider the input artifacts which we use. **Ensan et al.** [EBA+11] present a technique to select and prioritize product configurations based on the included features and their priority, which is defined by involved stakeholders. However, they do not prioritize specific test cases for product variants, but rather the product variants. **Lopez-Herrejon et al.** [LHJFC+14] propose a technique to generate prioritized pair-wise product variants. Thus, they combine pairwise sampling with a prioritization of product variants in an SPL. For prioritization, they compute weights for product variants. They solve the optimization problem using genetic algorithms. Compared to our work, they do not consider test models and do not prioritize test cases, but product variants. **Muccini** [Muc07] proposes a model differencing approach for test case selection based on single-software architectures. His approach provides ideas reused by our architecture analysis to identify changes between them. Basically, he focuses on structural and behavioral models and their similarities, which are represented in graph-like structures. **Parejo et al.** [PSS+16] devise a multi-objective test case prioritization approach for SPLs. While they describe their technique as test case prioritization, they assume that a test case resembles a product configuration. Thus, they define seven objectives for product variant prioritization, including a dissimilarity-objective and a fault history-based objective. In addition, they also analyze the size of features based on the underlying code. Using an multi-objective evolutionary algorithm, they observe that non-functional objectives outperform functional objectives in most cases for a real-world case study. In contrast to our work, they solely focus on product configurations. **Qu et al.** [QCR08] define a prioritization and sampling technique of configurations in configuration-aware systems which is, thus, applicable to SPLs. They apply a sampling technique for combinatorial interaction testing, leading to samples of configurations models. The generated samples are prioritized according to different metrics, such as block and fault coverage, which are based on code coverage metrics defined by Elbaum et al. [EMR01]. Compared to our work, they analyze configurations and their underlying code instead of changes in test models.

Test Case Related SPL Testing. Compared to the techniques described above, other SPL testing techniques (partially) focus on test cases instead of product variants. **Arrieta et al.** [AWSE16] define a test case prioritization approach for SPLs. In particular, their technique is specifically designed for cyber-physical systems. They apply weight-based search algorithms to prioritize test cases, using two types of optimization. The first metric is based on the execution costs of test cases. The second metric is based on the fault detection capabilities of test cases, which is similar to the test case history applied in our meta-data extension. Contrary to our work, they do not consider test models, delta-oriented testing nor integration test cases. **Baller et al.** [BL14, BLLS14] optimize SPL testing in a multi-objective fashion. Their test suite minimization and product variant prioritization approach is based on profit and coverage of test goals. They solve these problems using search-based techniques. In contrast to this thesis, they minimize the set of test cases for a product family, i.e., they compute product sub sets to optimize the cost to profit ratio. Furthermore, they do not prioritize test cases, but product variants, and are not using delta-oriented test models. **Knapp et al.** [KRS15] introduce a model-based test case selection technique for SPLs. They color test cases for product variants, i.e., they decide if they test specified behavior (green), unspecified behavior (yellow) or forbidden behavior (red) supporting the selection of test cases for product variants. In contrast to our testing framework for SPLs, their approach is based on model checking techniques and does not prioritize test cases. **Lackner** [Lac15] presents a model-based technique to sample product variants. He proposes coverage criteria to select a subset of product variants to increase fault detection capabilities, e.g., by focusing on small or diverse product variants. The technique uses reusable test cases as input and tries to execute each test case at least once, using sufficient product variants. Results of mutation testing indicate, that testing many products or small products has the highest positive impact on the fault detection of failures seeded in the variability model. However, the failure detection could not be increased for failures seeded in behavioral models. The sampling technique does not prioritize test cases as they focus in product variant sampling. **Lity et al.** [LMTS16] propose a test case selection technique for SPLs. Similar to our testing framework, they perform incremental testing of product variants using model-based regression testing approaches. They apply a slicing approach for delta-oriented state machines to identify important changes and their influences in product variants. In contrast to our work, they focus solely on unit testing, i.e., they do not consider any other test models such as architectures or MSCs. Additionally, they do not compute priority values for test cases. **Lochau et al.** [LLL+14] present a test case selection technique for integration testing of SPLs. Their incremental testing approach is based on delta-oriented architectures. Using delta-information, they analyze the current interface of components compared to the previously tested

variant. Their work presents one basis for our proposed framework, which takes these ideas further. In contrast to this thesis, they perform only a test case selection using only delta-oriented architectures. Their approach is not designed to be extended by other artifacts. Furthermore, they do not compute novel regression deltas, but only regression deltas to the successor of the current PUT. **Neto et al.** [DMSNdCMC+10] present a regression testing approach for SPLs which, similar to our work, uses software architectures as input data. Similarly, they perform a test case selection and prioritization of test cases defined as MSCs to maximize the reuse potential. In contrast to our work, they do not assess delta information or design an extensible framework. Their technique requires a high manual effort for the analysis of changes, which is a contrast to our testing framework. Furthermore, they require a reference architecture for the SPL as well as code, which we do not consider for our testing framework. **Reis et al.** [RMP07] propose a model-based testing technique for integration testing of SPLs. They analyze the control flow of a software system to derive interaction scenarios, which are the foundation for their test case generation. In contrast to our testing framework, they generate test cases in form activity diagrams without prioritizing them. **Uzuncaova et al.** [UKB10] present an test case generation approach for system-testing of SPLs based on specifications. The specifications are defined in first-order-logic. Similar to our work, they perform incremental testing of product variants. However, they do not perform delta-oriented testing nor do they perform model-based or regression testing approaches. **Varshosaz et al.** [VBM15] define a test case generation approach for delta-oriented SPLs. They derive state machines from delta-oriented code, allowing them to perform an efficient test case generation for product variants. While they also analyze deltas for their technique, they do not include any regression testing approaches in their work. **Wang et al.** [WAG13, WAG15] present a test suite minimization technique for SPLs. The propose several objectives to be optimized as once. Thus, they use weight-based genetic algorithms to solve their cost-cognizant test case minimization problem. Their objectives are based on the pairwise feature coverage, test minimization percentage and fault detection capabilities. In contrast to our work, their technique is search-based and does not focus on delta-oriented test models. In addition, they minimize test suites instead of prioritizing them.

Summarizing, the related work indicates that currently no technique with the same capabilities as our testing framework for SPLs exists. In particular, most existing techniques either focus on a selection or prioritization of product configurations instead of test cases or focus on test case generation and, thus, some form of test case selection for SPLs. In terms of integration testing, only few approaches exist with a similar scope. However, they are not defined to be easily extensible and often are not able to use the wide range of different input artifacts to prioritize test cases, which shows the necessity of our test case prioritization framework.

4.6 Chapter Summary and Future Work

Conclusion. SPL testing is a difficult problem due to a high number and the complexity of product variants. Thus, we propose a novel framework for integration testing of SPLs using delta-oriented test models. The output are prioritization functions applicable to test cases for certain product variants. We have defined functional and non-functional requirements for our testing framework to support the test case prioritization. We provided guidelines for the different steps in the framework to derive the foundations for a sophisticated test cases prioritization. The foundation of our test case prioritization framework are *weight metrics*, which are applicable to different artifacts. As we focus on integration testing, we compute model element weights. We provide sample weight metrics, e.g., based on changed architectural interfaces or multi product deltas. Using the weight metrics, our testing framework for SPLs supports the definition of *weight functions* and *test case prioritization functions* according to user needs and available input data. Our evaluation supports our claims and shows that both, the requirements are fulfilled and that the resulting test cases orders are of high quality. In addition, we show the extensibility of our test case prioritization framework for SPLs using different types of test models while incorporating additional black-box meta-data. An overview of related work shows the novelty of our testing framework for SPLs.

Future Work. Future work includes the evaluation with industrial large-scale SPLs with realistic failure data to avoid any bias. Also, the inclusion of code artifacts in the framework is of interest, i.e., how does implementation knowledge influence the test case prioritization and how can these artifacts properly included. Due to the extensibility of the framework, other data can be included as well, which shows a lot of potential for future applications of our test case prioritization approach. In addition, an adaption of the presented concepts into a search-based approach is interesting. Genetic algorithms have shown a large potential to find nearly optimal solutions in a reasonable amount of time [McM11]. Thus, the weight metrics defined using our framework can be used as objectives in a multi-objective optimization, resulting in Pareto-optimal solutions. This could lead to an automatic computation of the optimal weight factors, reducing manual effort.

5 Risk-Based Software Product Line Testing

The content of this chapter is mainly based on work published in [LBL+17].

Contribution

We contribute an approach to perform incremental risk-based testing for black-box software variants based on changes in test models. Our risk-based testing approach enables a test case prioritization for individual product variants, aiming to find important failures as early as possible. We compute impact values and failure probabilities for test model elements in a semi-automatic fashion, keeping manual effort low. The required manual assessments are independent of the number of tested product variants, providing scalability for large SPLs. We evaluate our risk-based testing technique, showing its applicability to integration testing of SPLs.

Risk-based testing (RBT) is a popular approach to evaluate single-software systems [ELR+14, FS14]. RBT is particular useful when resources for testing are sparse and important system parts are to be prioritized in testing. In RBT, risk metrics are applied to system artifacts to guide the testing process. Especially in safety-critical systems, different system parts yield different risk values, according to their importance and failure probability [Aml00]. However, one problem that current RBT approaches are not able to handle are large sets of product variants under test (PUT). Assume that an SPL should be tested, which comprises a large number of product variants. Testing each product variant is infeasible. Instead, important parts of each variant shall be identified automatically based on a notion of *risk*. Manually assigning risk values for each product variant is infeasible, especially if, e.g., components of product variants shall be assessed separately.

Currently, only Hartmann et al. [HvdLB14] present a risk-based testing approach for SPLs. They use quantified feature models, i.e., each feature is assigned a *failure probability* and *failure impact* value. The resulting values help to identify important features and, thus, product variants. However, this approach only scales on product variant level, e.g., using sampling methods.

To fill this gap, this chapter introduces an efficient risk-based testing approach for SPLs with the goal to *prioritize test cases* for SPLs with similar intentions as our SPL framework (cf. Chapter 4). We introduce a model-based approach for risk-based testing based on delta knowledge [LBL+17]. It analyzes differences between

product variants to identify parts with high risk values. Compared to traditional single software system approaches, our RBT technique is able to compute risk values for product variant parts automatically. In other words, it computes risk values for different test model elements for a certain product variant. Test cases which cover risky system parts are more important in their execution and, thus, receive a higher priority value. Our RBT approach for software variants only requires an initial manual setup, which is independent of the number of *product variants under test* (PUT). The computed risk values enable an automatic prioritization of test cases, with the goal to find important failures as early as possible.

Our evaluation shows the potential of our risk-based test case prioritization technique in terms of failure finding rate. As evaluation scenario, we apply our RBT approach to integration testing of SPLs. In addition, we are able to show the integration into our SPL framework (cf. Chapter 4). We evaluate our risk-based SPL testing technique in isolation as well as in comparison with our delta-oriented testing approach for a more in-depth assessment.

In the remainder of this chapter, we first give a short introduction into risk-based testing. Next, we describe the details of our risk-based SPL testing technique and how we are able to efficiently compute risk values for SPLs. We perform our evaluation with data provided by BCS. Our evaluation indicates that our RBT approach is able to effectively detect important failures in integration testing. It improves results compared to our delta-oriented test case prioritization approach and a random test case prioritization. Finally, we present related work close to our RBT technique for software variants and conclude the chapter with future work.

5.1 Risk-Based Testing

Resources are often limited in software testing, which increases the need for efficient and effective regression testing techniques [ERS10]. Thus, it is important to focus testing effort on important parts of the system, i.e., parts which are likely to fail or whose failings have a high impact. One approach to identify important parts is to apply *risk-based testing* (RBT) [Aml00, FS14]. In RBT, risk describes the likelihood that something unintended will happen with a negative impact to the system. Thus, risk metrics can be defined for different types of artifacts in software engineering, e.g., requirements or components. Amland [Aml00] provides the standard definition of risk based on two factors: the *failure probability* and the *failure impact*. In other words, the more likely a system part is to fail and the worse the consequences a failure causes, the higher is the risk of this particular system part.

We are able to rank the importance of system parts, e.g., components or classes, according to their risk values. Starting with the part with the highest risk value, the testing process ensures that the most important parts are tested first.

Definition 5.1: Definition of Risk in Software Engineering

Based on the risk definition by Amland [Aml00], we define the risk of an artifact a based on the artifact's failure impact $f_{impact}(a)$ and failure probability $f_{prob}(a)$ as follows:

$$risk(a) = f_{impact}(a) \cdot f_{prob}(a)$$

The most important and difficult part of RBT is the derivation of risk values. Especially in the beginning of the software engineering process, risk assessment is performed manually by experts and stakeholders, which have in-depth knowledge about the different parts of the system under development [ELR+14]. Automatic risk assessments are often performed by static analysis tools, which require code access [FS14]. In this thesis, we assume to neither have code access nor to analyze single-software systems.

5.2 Efficient Risk-Based Testing Technique for SPLs

We propose a novel black-box test case prioritization approach for SPLs based on the notion of risk. Our goal is to reduce the manual effort compared to traditional techniques and exploit the reuse potential of SPLs in an incremental testing process. An overview of our RBT approach is shown in Figure 5.1.

We distinguish two major phases to perform RBT for SPLs. The first phase is part of the domain engineering of an SPL and sets the testing process up. This phase provides the setup for our RBT technique and is performed manually. The second phase is the incremental test case prioritization for each PUT, which is performed completely automatically. Here, test cases are prioritized in an incremental fashion for one product variant after the next. For each product variant, we analyze risk values for parts of the given test model and compute test case priorities according to a test case's covered test model elements. We explain both phases and their respective subphases in the following.

5.2.1 Domain Engineering and Preparation

In the first major phase of our risk-based test case prioritization approach, the SPL has to be defined and necessary artifacts have to be designed in a model-based fashion. This also comprises a mapping from each part of a test model to the corresponding features for the core variant of the SPL. In total, we distinguish three subphases (cf. left-hand side of Figure 5.1).

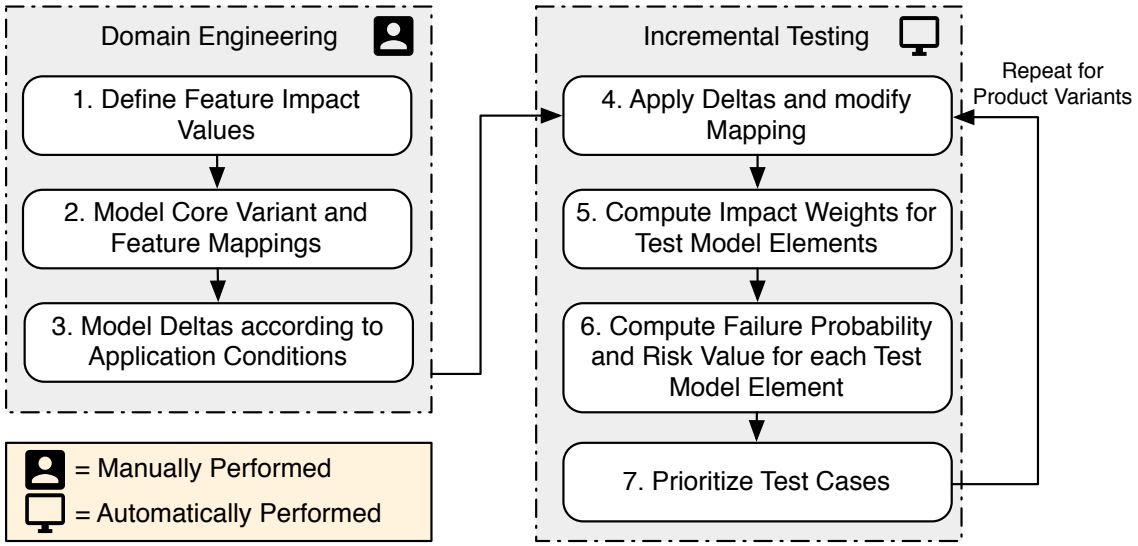

Figure 5.1: Risk-Based Testing Approach for SPLs (cf. [LBL+17])

1. Define Feature Impact Values.

In the first step of our RBT approach for software variants, similar to Hartmann et al. [HvdLB14], we use quantified feature models (QFM) to specify a *feature impact* value for each non-abstract feature. Our goal is to use these values later to compute feature impact values for test model elements. QFM support the assignment of variables for features in the feature model. In context of this thesis, assigned numerical values represent the importance of particular, non-abstract features. For instance, a feature *security* might have a value of 5 assigned, while a feature *log* only has value of 2, indicating less importance in case of failure. If an important feature fails, the system is likely to produce a costly failure. In contrast to Hartmann et al. [HvdLB14], we do not require a manual assignment of *feature failure probability* values as they are computed automatically in the second phase of our RBT technique. This reduces the manual effort of RBT for SPLs. However, we argue that a manual assessment of feature impact values is important as it requires expert knowledge. By assigning impact values to features we avoid that an expert has to impact values for each product variant under test separately, which is infeasible.

A *feature impact value* $FI : F_{SPL} \rightarrow [1,5] = \{x \in \mathbb{N} \| \ 1 \leq x \leq 5\}$ has to be defined for each feature $f \in F_{SPL}$, where F_{SPL} denotes the set of all features for the SPL. We define feature impact values to be on a fixed scale from 1 to 5, where 5 represents the highest possible impact in case of a failure. However, we do not restrict our RBT approach to this scale, but give an indication for typical values.

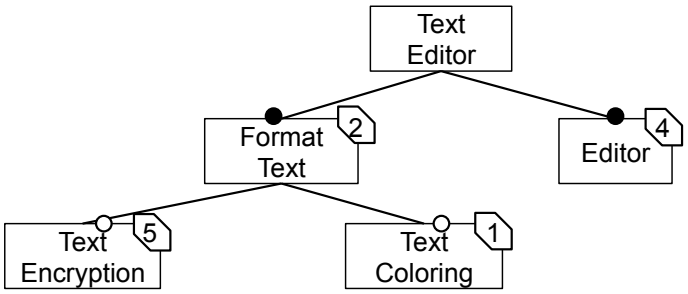

Figure 5.2: QFM with Feature Impact Values

Example 5.1: QFM with Impact Values

We present a sample QFM for a simple text editor SPL with feature impact values in Figure 5.2. The QFM contains five features: Text Editor, which represents the abstract root feature, a mandatory Editor and the mandatory feature Format Text. The latter has two optional child features: Text Encryption and Text Coloring. All features but Text Editor (abstract) have a feature impact value assigned as follows:

$FI(\text{Text Coloring}) = 1$, $FI(\text{Format Text}) = 2$, $FI(\text{Editor}) = 4$ and $FI(\text{Text Encryption}) = 5$.

The Text Encryption has a high impact value as it represents a security related feature. In contrast, the Text Coloring feature is of the lowest impact as it is only a design-related feature.

We define a function $FI_{F_{SPL}} : \mathcal{P}(F_{SPL}) \rightarrow \mathbb{N}$ (cf. Definition 5.2) to compute the sum of all feature impact values for a set of features $FS \subseteq F_{SPL}$. This is important for later risk computations for test model elements of specific product variants.

Definition 5.2: Feature Set Impact Values

Let F_{SPL} be the set of features and $FI : F_{SPL} \rightarrow [1,5] = \{x \in \mathbb{N} \| \ 1 \leq x \leq 5\}$ the feature impact value function.

We define a *feature set impact value function* $FI_{F_{SPL}} : \mathcal{P}(F_{SPL}) \rightarrow \mathbb{N}$ which returns the sum of feature impact values for a set of features $FS \subseteq F_{SPL}$ as:

$$FI_{F_{SPL}}(FS) = \sum_{i=1}^{|FS|} FI(f_i)$$

In the context of this thesis, we do not focus on techniques to retrieve or generate feature impact values. Instead, we assume that an expert is able to perform this task manually, which is similar to currently performed risk assessments. One advantage is, that this task has to be performed only once for an SPL. Updates are only necessary, if changes in features occur or previous assessments of feature impact values are no longer valid and have to be updated.

2. Model Core Variant and Feature Mappings.

For our RBT approach for black-box software variants, we assume to have knowledge about a *test model* which represents the current PUT on a suitable granularity, according to the test focus. For example, component testing should provide state-based models and integration testing should provide architectural test models. In the context of this thesis, we do not specialize on a certain type of test model, but present a generic approach applicable to different test models.

In the remainder of this chapter, we use the terminology introduced in Chapter 3.1.1 to describe our RBT approach in an abstract, flexible fashion: Test models are represented as a finite non-empty set of *vertices* $V = \{v_1, \ldots, v_n\}$ and *edges* $E = \{e_1, \ldots, e_m\}$, where each edge $e = (v, v')$ connects two vertices in a directed way from v to v' (cf. abstract test model definition in Chapter 3.1.1). The set of vertices and edges are aggregated as the set of *test model elements* $\Omega = \{\omega_1, \ldots, \omega_k\} = V \cup E$. As described in Chapter 3.1.1, a vertex describes a certain state or responsibility in the test model specification of the system. The specific semantics of a vertex depend on the type of test model, e.g., in behavioral test models, it represents a state the system is in, while it represents a component or system part in structural test models. Each vertex has to be reachable, i.e., the underlying graph is connected via its edges.

To perform our risk-based SPL testing approach, a core variant has to be specified [CHS10]. Once a core is selected, experts have to manually map the vertices and transitions of the core test model to features in the QFM. This step is required as we later use feature impact values to analyze these elements and compute risk values for vertices. Smaller core variants lead to a lower effort as they require less mappings to be performed.

Independent of the core size or its specifics, feature mappings for each test model element $\omega \in \Omega$ have to be provided. Thus, we define a function $F_{core} : \Omega \times P_{SPL} \rightarrow \mathcal{P}(F_{SPL})$, which returns the set of features mapped to a test model element $\omega \in \Omega$ of the core variant. For example, for integration testing feature mappings for each component $c \in C$ (representing the vertices) and each connector $con \in CON$ (representing the edges) have to be provided for the core variant. A more complex example is described in Example 5.2.

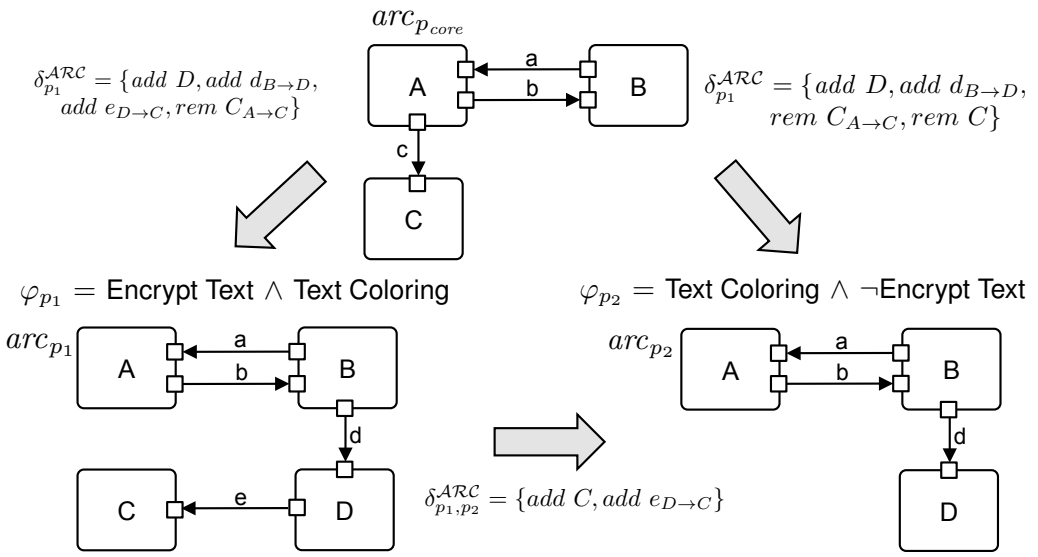

Figure 5.3: Sample Product Variants for Risk Computation on Architecture Level

Example 5.2: Core Mappings for Architecture Models

Figure 5.3 shows three product variants derived from the SPL defined by the QFM shown in Example 5.1. We use these product variants as product variants under test (PUTs) for following examples. In our running example, we focus on integration testing, i.e., the product variants are described via architecture test models. The goal of this chapter is to be able to provide risk values for each test model vertex in all product variants under test, i.e., each component in the running example. In addition, we aim to require minimal manual effort and perform this computation in an automatic fashion.

Thus, we require to map elements from the core variant's architecture $arc_{p_{core}}$ to the features provided in Figure 5.2. The core variant is defined for the mandatory features of the SPL. Assume the following sample mappings:

$F_{core}(A, p_{core}) = F_{core}(C, p_{core}) = F_{core}(c, p_{core}) =$ Format Text
$F_{core}(B, p_{core}) = F_{core}(a, p_{core}) = F_{core}(b, p_{core}) =$ Editor

3. Model Deltas according to Application Conditions.

The user has to specify a set of deltas to represent the variability of the SPL [CHS10]. Deltas are applied to the core variant to generate other product variants. We use

these deltas to identify important changes in product variants under test. The function *operations* : $\Omega \times P_{SPL} \rightarrow \mathcal{OP}$ returns the set of delta operations $op \in \mathcal{OP}$ that correspond to an element in the system, e.g., by adding or removing it (cf. Definition 3.4 in Chapter 3.1.1). For integration testing, for instance, delta operations influence components and connectors in the system.

As defined in Chapter 3.1.1, deltas are applied according to their application condition φ. In the context of this thesis, they are defined as boolean formula over the set of features F_{SPL}, denoted as $\mathbb{B}(F_{SPL})$. If an application condition is evaluated to `true`, the corresponding delta is applied for the generation of a product variant. We later analyze application conditions to automatically update mappings for other product variants. Sample application conditions are shown in Figure 5.3.

5.2.2 Risk-Based Incremental Testing

After an SPL has been designed and core variant mappings are provided, incremental risk-based testing can commence (cf. right-hand side of Figure 5.1). In this part of our risk-based test case prioritization technique for black-box software variants, no further manual effort is required. Our main idea is to automatically compute risk values for each vertex in a test model for each product variant under test. Based on these risk values, we are able to prioritize test cases. We explain the four corresponding subphases in the following.

4. Apply Deltas and modify Mappings.

The first step in our RBT approach for software variants is the derivation of the product variants under test (PUT) (cf. Figure 5.1). To generate test models for each PUT, we apply the required deltas to the core. The application conditions allow us to map parts of the test model of the PUT to corresponding SPL features. To this end, we introduce the *mapping function* $F_{map} : \Omega \times P_{SPL} \times \Delta_{SPL} \rightarrow \mathcal{P}(F_{SPL})$, which returns the set of mapped features for a test model part $\omega \in \Omega$. This function only returns features, which are present in the current product configuration, i.e., $f \in FC_p$ (cf. Chapter 2.3.2) for product variant p. This excludes any manual mappings made for elements of the core variant, which are still present in the current PUT, i.e., it only returns changed mappings influenced by applied deltas. This is to avoid any confusion with the mappings manually created for the core variant as explained in Step 2. The result are mappings for a test model element $\omega \in \Omega$, either from the core or variant-specific.

Definition 5.3: Feature Mapping Function

Let Ω be the set of test model elements, P_{SPL} the set of product variants, Δ_p the set of deltas applied to retrieve product variant p, $conditions : \mathbb{B}(F_{SPL}) \rightarrow \mathcal{P}(F_{SPL})$ a function which returns the features in a delta's application condition and $F_{core} : \Omega \times P_{SPL} \rightarrow \mathcal{P}(F_{SPL})$ the mapping function for the core variant, retrieving features for all test model elements in the core test model.

We define the *feature mapping function* $F_{map} : \Omega \times P_{SPL} \times \Delta \rightarrow \mathcal{P}(F_{SPL})$, which retrieves the set of features mapped to a test model element $\omega \in \Omega$ in product variant $p \in P_{SPL}$ as follows:

$$F_{map}(\omega, p, \Delta_p) = F_{core}(\omega, p) \cup \left(\bigcup_{j=1}^{|\Delta_p|} conditions(\Delta_{p_j}) \right)$$

With the feature mapping function, we are able to automatically retrieve mappings for an arbitrary element in the test model of the current PUT. This reduces manual effort in SPL testing. Example 5.3 shows the mappings computed for our running example in terms of integration testing.

Example 5.3: Automatic Mapping Computation

By applying the feature mapping function to the test model elements in product variant p_2 using the corresponding deltas Δ_{p_2}, we obtain the following feature to element mappings for test model elements:
$F_{map}(A, p_2, \Delta_{p_2}) =$ Format Text
$F_{map}(B, p_2, \Delta_{p_2}) = F_{map}(a, p_2, \Delta_{p_2}) = F_{map}(b, p_2, \Delta_{p_2}) =$ Editor
$F_{map}(D, p_2, \Delta_{p_2}) = F_{map}(d, p_2, \Delta_{p_2}) =$ Text Coloring

5. Compute Impact Weights for Test Model Elements.

Risk-based testing is based on the notion of risk. Thus, we propose a novel way to compute risk values for test model *vertices* $v \in V$ in the graph representation of the product variant specification, e.g., states in state machines or components in architectures. We do not compute risk values for edges as vertices represent states or parts of the system, which are responsible for a system's behavior. To avoid manual assignment of risk values in SPLs, we compute risk values automatically based on the feature impact values defined earlier as well as the relationships of test model elements and the number of their occurrences in tested product variants.

Risk is defined as the product of impact values and failure probability. In this phase, we compute the *impact value* for a vertex v based on the feature mappings for each element in the test model. Vertex impact values are based on three different influences, explained in the following:

Vertex Feature Impact Values: First, the impact values of the mapped features for vertex v are important. If a vertex is mapped to an important feature of the SPL, we argue that testing the underlying functionality of this vertex is important as well. Thus, the risk value of a vertex v in product variant p depends on the features values of the features returned by the feature mapping function $F_{map}(v, p, \Delta_p)$.

Vertex Neighborhood: The second influence are vertices in the *neighborhood* of a vertex $v \in V$, i.e., vertices which are connected to v via at least one edge. This is important, as the communication of one risky vertex with another might influence the risk potential of both. We use the vertex neighborhood function $NC : V \times P_{SPL} \rightarrow \mathcal{P}(V)$ to identify the neighbors of a vertex $v \in V$ (cf. Definition 3.2 in Chapter 3.1.1). Thereupon, we compute the features mapped to these vertices via $F_{map}(v', p, \Delta_p)$ for all vertices $v' \in NC(v, p)$. Their feature impact values are later used to compute the vertex impact value.

Interface Feature Impact Values: The third influence for vertex impact values are a vertex's interface, i.e., its incoming and outgoing edges in the test model. The interface of a vertex is returned by the *vertex interface function Int* : $V \times P_{SPL} \rightarrow \mathcal{P}(E)$ as defined in Definition 3.3 in Chapter 3.1.1. It contains all incoming and outgoing edges connected to vertex v in product variant p. Based on the interface of a vertex v, we are able to retrieve the features mapped to the edges $e \in Int(v, p)$. These mapping are returned using the feature mapping function $F_{map}(e, p, \Delta_p)$.

We compute the feature impact values for all extracted features for a vertex v and aggregate them into one *vertex impact value*. We define a *vertex impact function* to compute the vertex impact as $impact : V \times P_{SPL} \times \Delta_{SPL} \rightarrow [0, 1]$ [LBL+17]. It computes the sum of all feature impact values of mapped features for the vertex $v \in V$, its neighbor vertices and its interfaces edges.

The vertex impact value is normalized by the maximum feature impact value times the number of mapped features. This avoids that one large feature impact value or a large interface and neighborhood automatically increases the impact value, while vertices with small interfaces and less neighbors are less important. This leads in a resulting value between 0 and 1 for each vertex, which indicates its importance in the system. Consequently, test cases which cover important vertices should be executed with higher priority.

Definition 5.4: Vertex Impact Function

Let V be the set of vertices, Δ_{SPL} the set of deltas, $\omega \in \Omega$ a test model element, F_{SPL} the set of features of the SPL, $NC : V \times P_{SPL} \rightarrow \mathcal{P}(V)$ the function of neighbor vertices of a vertex, $Int : V \times P_{SPL} \rightarrow \mathcal{P}(E)$ the interface function returning the set of edges in the interface of a vertex v, $F_{map} : \Omega \times P_{SPL} \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(F_{SPL})$ the feature mapping function returning the features mapped for an model element in the product variant and $FI_{F_{SPL}} : \mathcal{P}(F_{SPL}) \rightarrow \mathbb{N}$ a function to compute the sum of feature impact values for a a set of features.

We define the vertex impact function $impact : V \times P_{SPL} \times \mathcal{P}(\Delta_{SPL}) \rightarrow [0, 1]$ to compute the vertex impact value for each vertex $v \in V_p$ in a product variant $p \in P_{SPL}$ based on the set of deltas Δ_p applied for p as follows:

$$impact(v, p, \Delta_p) = \frac{FI_{F_{SPL}}(\bigcup F_{map}(\omega, p, \Delta_p))}{|\bigcup F_{map}(\omega, p, \Delta_p)| \cdot max(\bigcup_{k=1}^{|F_{SPL}|} FI_{F_{SPL}}(f_k))}$$

with $\omega \in (v \cup Int(v, p) \cup NC(v, p))$.

Example 5.4: Component Impact Value Computation

Based on our running example, we compute the component impact value for component B in product variant p_2:

$$impact(B, p_2, \Delta_{p_2}) = \frac{FI_{F_{SPL}}(\{\text{Editor, Text Format, Text Coloring}\})}{|\{\text{Editor, Text Format, Text Coloring}\}| \cdot FI_{F_{SPL}}(\text{Text Encryption})} = \frac{7}{3 \cdot 5} \approx 0.47$$

6. Compute Failure Probability and Risk Value for each Vertex.

The second factor for our risk computation is the *failure probability* of a vertex. Again, our goal is to avoid a manual assignment of failure probability values in SPLs, which does not scale for a larger number of product variants. Instead, we propose a novel way to compute failure probabilities for test model vertices automatically. In particular, we analyze the occurrences of a certain test model vertex $v \in V$ in the set of previously tested variants, denoted as P_v. The higher the number of occurrences, the higher is the likelihood that the vertex and its represented functionality has been tested sufficiently, which reduces the failure probability. One advantage of this assumption is that the failure probability has not to be defined manually by the

test expert, which is a difficult task and becomes infeasible for a large number of product variants.

The computation of a vertex's failure probability in product variant p based on the set of previously tested variant $P_{tested} \subseteq P_{SPL}$ is described as function *fprob* : $V \times P_{SPL} \times \mathcal{P}(P_{SPL}) \rightarrow [0, 1]$. The highest failure probability value is 1, which, in contrast to probability theory, does not indicate that a failure necessarily occurs as it is impossible to predict a priori if a failure will actually occur. Instead, it indicates a high importance for retesting of this vertex in terms of failure probability.

First, the failure probability function assesses the delta operations $OP_{v,p} = \{op_1, \ldots, op_k\}$ in the current PUT $p \in P_{SPL}$ which are related to vertex v. The corresponding delta operations are identified using the function *operations* : $\Omega \times P_{SPL} \rightarrow \mathcal{P}(\mathcal{OP})$, which returns the set of delta operations for a test model element $\omega \in \Omega$ (cf. Chapter 3.1.1). Next, the function computes the average number of symmetrical differences ($\oplus$) of the current delta operations related to vertex $v \in V$ to the delta operations for previously tested variants, in which vertex v also occurred, denoted as $P_v \subseteq P_{tested}$. This is similar to the computation of regression deltas as described in Chapter 3.1.1. The result of the symmetrical difference are delta operations, which influence the interface of vertex v compared to previously tested variants. Thus, the symmetrical difference of the delta operations indicate how similar the previously tested interfaces of a test model vertex v are to the current variant of the same vertex. The more elements are in the set retrieved by the symmetrical differences, the less the current configuration of v has been tested. This increases the failure probability of the currently tested variant of a vertex, as previously tested configurations of v have been largely different from the current variant. This reduces the significance of previously gathered test results for the particular vertex in the current product variant.

We normalize the failure probability computation for vertex $v \in V$ by the number of products in which v already occurred and the maximum difference between the current variant of v and previously tested variants of v in the set of of product variants $P_v \subseteq P_{tested}$, which also contain v. This leads to a failure probability value between 0 and 1, where 1 indicates a high failure probability. Again, a value of 1 does not indicate that a failure happens. This kind of prediction is not possible and, thus, the failure probability value is only an indication of importance, but does not predict a definitive failing of a system part. Our computation reduces the failure probability with an increasing number of occurrences as we assume that the vertex has been tested more often in previous product variants. This is resembles current testing trends as continuous testing increases the trust into the system and its correct implementation.

Definition 5.5: Vertex Failure Probability

Let $P_v \subseteq P_{SPL}$ be the set of tested product variants which contain vertex $v \in V$, $OP_{v,p}$ the set of delta operations related to vertex v in product variant p and $\oplus$ the symmetric difference operator.

We define the *vertex failure probability* function $fprob : V \times P_{SPL} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$ for a vertex $v \in V_p$ in product variant $p \in P_{SPL}$ based on the previously tested product variants P_v containing v as follows:

$$fprob(v, p, P_v) = \begin{cases} \frac{\sum_{k=1}^{|P_v|} |OP_{v,p} \oplus OP_{v,p_k}|}{|P_v| \cdot max(\bigcup_{l=1}^{|P_v|} |OP_{v,p} \oplus OP_{v,p_l}|)} & if\ (|P_v| \cdot max(\bigcup_{l=1}^{|P_v|} |OP_{v,p} \oplus OP_{v,p_l}|)) > 0 \\ 0 & otherwise \end{cases}$$

Example 5.5: Component Failure Probability Computation

Based on Example 5.2, we are able to compute the failure probability for component B in product variants p_2 as follows:

$$fprob(B, p_2, P_B) = \frac{1}{2 \cdot 1} = 0.5$$

Both, the impact value and the failure probability are multiplied to a risk value for each vertex $v \in V_p$ for a test model of product variant p, similar to risk in single-software systems [Aml00]. This results in the risk function $risk : V \times P_{SPL} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$. Using this function, we are able to compute risk values for each vertex in a product variant test model. This makes RBT for SPLs feasible.

Definition 5.6: Vertex Risk Computation

Let $P_v \subseteq P_{tested}$ be the set of tested product variants containing vertex $v \in V$, $V_p \subseteq V$ the set of vertices for a test model of product variant $p \in P_{SPL}$, $impact : V \times P_{SPL} \to \mathbb{R}$ a failure impact function and $fprob : V \times P_{SPL} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$ a failure probability function.

We define the *vertex risk function* $risk : V \times P_{SPL} \times \mathcal{P}(P_{SPL}) \to \mathbb{R}$ for a vertex $v \in V_p$ for product variant $p \in P_{SPL}$ and set of previously tested variants P_v containing vertex v as follows:

$$risk(v, p, P_v) = impact(v, p) \cdot fprob(v, p, P_v)$$

Example 5.6: Component Risk Value Computation

Based on the values derived in Example 5.4 and Example 5.5, we are able to compute the risk value for component B in product variant p_2 as follows:

$$risk(B, p_2, P_B) = 0.47 \cdot 0.5 = 0.235$$

The value is rather low and, thus, integration testing of communication-scenarios of this particular component might be less important.

7. Prioritize Test Cases.

We compute priority values for a set of test cases $\mathcal{TC} = \{tc_1, \ldots, tc_n\}$ based on the computed risk values for vertices in test models. The higher the priority value, the more important is a test case and the earlier it is executed. Test cases have to be provided as we do not focus on test case generation in this thesis. In addition, test cases have to fit the provided test models, i.e., they have to correspond to the test models and the current testing phase. For example, integration testing requires test cases which assess the communication of components, e.g., defined as message sequence charts (cf. Chapter 3.1.1). We do only prioritize test cases, which are applicable to the current PUT and are not new as new test cases have to be always executed to ensure that new behavior is tested.

Similar to the test case prioritization functions shown in Chapter 4, we define a test case prioritization function $prio_{risk} : \mathcal{TC} \times P_{SPL} \to \mathbb{R}$, which assigns a priority value to a given test case $tc \in \mathcal{TC}$, which is applicable to product variant $p \in P_{SPL}$. The priority is based on the computed risk values for vertices. Risk is incorporated in such a way, that the function assesses each tested edge between two vertices in the test model and computes the risk of the involved vertices. In other words, we investigate vertex and edge coverage. We do not compute risk values for edges, as their mapped feature impact values are analyzed for the vertex weights. We assume that a different number of occurrences of edges leads to different test cases. Thus, we compute the multi-set of covered edges between two vertices using the function $cov_e : V \times V \to \mathcal{P}(E)$. Consequently, $cov_e(v, v')((v, v'))$ returns the multiplicity of occurrences of edge $(v, v') = e$. This approach resembles the idea of the communication-based test case prioritization in our SPL framework (cf. Chapter 4.4.3). For each covered edge, we analyze if involved vertices are of high risk. In this case, corresponding test cases shall also be of high importance. The more important vertices are traversed by a test case, the higher is the importance of the test case. The resulting test case prioritization function is normalized by the number of tested edges. This reduces the chance that very complex test cases are automatically of high importance.

Definition 5.7: Risk-Based Test Case Prioritization

Let $V_p \subseteq V$ be the finite set of vertices in product variant $p \in P_{SPL}$, $V_{tc} \subseteq V_p$ the finite set of vertices covered by test case $tc \in \mathcal{TC}$, $cov_e : V \times V \to \mathcal{P}(E)$ the multi-set of edges between two specific vertices in a test case $tc \in \mathcal{TC}$ and $risk : V \times P_{SPL} \times \mathcal{P}(V) \to \mathbb{R}$ a vertex risk function.

We define the *risk-based test case prioritization function* $prio_{risk} : \mathcal{TC} \times P_{SPL} \to \mathbb{R}$ for a test case $tc \in \mathcal{TC}$ for a particular product variant $p \in P_{SPL}$ based on the risk values of covered test model vertices as follows:

$$prio_{risk}(tc, p) = \frac{\sum_{n=1}^{|V_{tc}|} \sum_{m=n+1}^{|V_{tc}|} \{k \cdot (risk(v_n, p, P_{v_n}) + risk(v_m, p, P_{v_m})) | cov_e(v_n, v_m)((v_n, v_m)) = k\}}{\sum_{n=1}^{|V_{tc}|} \sum_{m=n+1}^{|V_{tc}|} |cov_e(v_n, v_m)|}$$

Test cases are executed according to their priority. Testing is repeated for each product variant under test. Due to the incremental nature of our RBT approach, the priority values of test cases may differ from product variant to product variant. This enables a flexible test case prioritization for individual product variants.

5.3 Evaluation

We evaluate our black-box RBT technique for SPLs in the context of integration testing. For our evaluation, we first design research questions to be answered. Next, we explain our evaluation methodology. We use 17 product variants of BCS as subject system as BCS provides the necessary model-based artifacts (cf. Chapter 3.2). The product variants are the ones described in Chapter 3.2. We show and discuss our results and explain potential threats to validity.

5.3.1 Research Questions

RQ1: *How* effective *is the risk-based testing technique in terms of failure finding rate?* Our main goal is to find important failures as early as possible. Thus, we analyze if our RBT technique is able to outperform other approaches in terms of failure finding rate. In addition, we assess if it can be integrated in our previously described SPL test case prioritization framework.

RQ2: *How* efficient *is our RBT technique for software variants in comparison to manual risk-based testing approaches, i.e., does it scale for SPLs?* We investigate if our RBT technique scales for SPLs, i.e., if it applicable to a set of

product variants without increasing the manual testing effort. Furthermore, we investigate the necessary manual effort compared to the automatically performed actions and how much time our risk value computation takes.

5.3.2 Methodology

We used our expert knowledge about the system to manually assign feature impact values for 27 features of BCS [LBL+17]. For our evaluation, we focus on integration testing as BCS offers all necessary artifacts for this testing phase and we are able to compare results to our SPL framework [LLLS12, LSN+16, LBL+17]. We use architectural test models for our evaluation, which consists of components and connectors as explained in Chapter 3.1.1. Thus, the concept of *vertices* is now implemented by components and *edges* are now represented by connectors. We map features to components and connectors of the core architecture test model.

To answer RQ1, we investigate the effectiveness of our risk-based test case prioritization technique for SPLs by measuring the *average percentage of faults detected* (APFD) metric [RUCH01] (cf. Chapter 2.2.2). In this thesis, APFD allows us to assess how fast failures are covered. As BCS does not provide failure information, we perform mutation testing [JH11] to assess the failure finding rate. Here, APFD indicates how fast we are able to cover the seeded failures. In total, two different types of failure seedings are performed to capture different types of changes.

Experiment 1. First, we seed failures in connectors that correspond to changed components in a product variant. This resembles our assumption that failures occur in changes. Thus, we seed failures in interfaces which correlate to changed components as we focus on integration testing. Failures are newly seeded for each product variant. To reduce the number of seeded failures per component, we compute the change ratio for the component in the current product variant. A maximum of 10% of interface connectors can be faulty to avoid failure masking, i.e., failures could avoid the detection of other failures. In addition, too many failures per product variant are not realistic and make the failure detection easy. For each potential connector, the change ratio of the component dictates the chance that a failure is actually seeded. In other words, the higher the change ratio of the component to previous variants is, the higher is the likelihood that a connector contains a fault. To prevent statistical outliers, we set up a fixed upper limit of 5 failures per component. This procedure reduces the number of actual failures per product variant to an average of 1.5. This is similar to the assumption of mutation testing, where only first order mutants are seeded, i.e., simple failures which represent failures made by human expert [DLS78]. A low number of failures makes it hard to find failures early, which further proves the potential of our test case prioritization approach.

Experiment 2. While Experiment 1 focuses on failures in changed interfaces, our main goal is to detect *important* failures early on. Thus, we refine the seeding of Experiment 1 to create failures in connectors, which are part of changed component's interfaces and additionally correspond to features in the QFM with a feature impact of 3 or higher. This enables us to investigate whether our test case prioritization technique is able to detect important failures in product variants early, which would cause a high impact on the system. All other failure seeding characteristics are the same as in Experiment 1.

Experiment Repetition. As we perform mutation testing, we repeat our experiments 100 times to avoid statistical outliers and anomalies. We aggregate APFD results to show average values for all experiment repetitions. Random testing is performed as baseline comparison, which is also repeated 100 times.

5.3.3 Results and Discussion

> **RQ1:** *How* effective *is the risk-based testing technique in terms of failure finding rate?*

Compared Techniques and Integration into SPL Framework. We assess the effectiveness of our RBT approach for software variants by measuring the APFD metric [RUCH01]. In total, four test case prioritization techniques are compared: *Random Prioritization*, *Delta-Oriented Prioritization*, *Risk-Based Prioritization* and a combination of the latter two. We are able to combine the two approaches due to their similar structure and the usage of our SPL testing framework. In particular, we are able to introduce a risk-based weight metric in our SPL testing framework. Thus, the computed risk values for components is integrated as new weight metric, as it provides weights for particular components in the system, similar to our other component-based weight metrics. Furthermore, this risk-based weight metric is part of an adapted test case prioritization function based on the component-based delta-oriented test case prioritization approach. Using weighting factors, we are able to adjust the impact of the risk-based weight metric in the overall prioritization function. Thus, we achieve that both, risk-based testing and delta-oriented testing are applied simultaneously. We are able to regulate the impact of either approach using weight factors as defined in Chapter 4.2.3.

Results of Experiment 1. The results of the four test case prioritization approaches, shown in Figure 5.4, indicate that the random test case prioritization technique is significantly outperformed by the other three techniques with a median APFD of 0.72. The best result is achieved by the delta-oriented approach (*RegDelta*) with a median APFD value of 0.86. Our risk-based test case prioritization approach

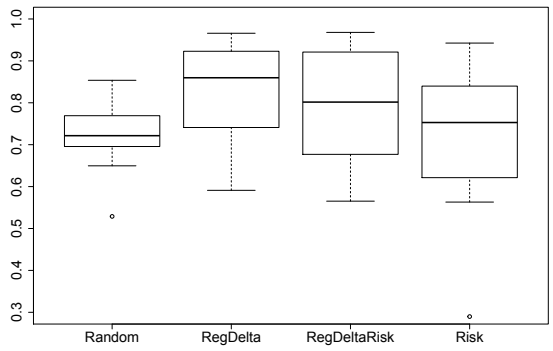

Figure 5.4: Boxplots for Results of Experiment 1 (cf. [LBL+17])

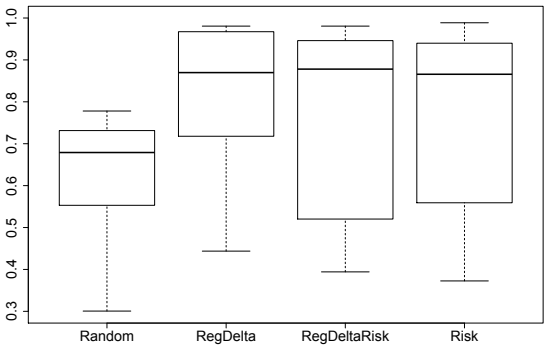

Figure 5.5: Boxplots for Results of Experiment 2 (cf. [LBL+17])

achieves a median APFD of 0.75 (*Risk*). While this is already a good result, it does not outperform the existing delta-oriented technique. The combination of both (*RegDeltaRisk*) results in an APFD in between compared to an execution of both techniques in separation. The results are explained by the failure seeding, i.e., we seed failures in arbitrary interfaces of changed components, which is covered well by the delta-oriented approach. However, the risk-based approach prioritizes important parts of the system first, which, in this experiment, do not necessarily yield failures.

Results of Experiment 2. To assess the quality of our RBT technique for software variants in terms of important failures, we present the results of the Experiment 2 in Figure 5.5. Again, the average results of the four techniques are shown as boxplots. The random prioritization technique still performs worst with a median APFD of 0.67. However, the risk-based technique has improved significantly compared to Experiment 1, now achieving a median APFD of 0.86. We expected an improvement, but this very high APFD value shows the potential of our black-box RBT technique to focus on important parts of the system and to reveal important failures early. The delta-oriented technique still achieves a good median APFD value of 0.84. The combination of the risk-based and delta-oriented technique outperforms both. As a result, it achieves the highest APFD median of 0.87. This is due to the combination of both assumptions, failures in changed interfaces and failures in important changes.

We are able to deduce that our risk-based test case prioritization technique for software variants is able to find important failures and in average outperforms existing techniques such as the delta-oriented technique. This validates our first research question. Furthermore, we show that the risk-based approach can be integrated into our framework. This increases the failure finding potential even further.

RQ2: *How* efficient *is our RBT technique for software variants in comparison to manual risk-based testing approaches, i.e., does it scale for SPLs?*

In addition to the effectiveness of our RBT technique, we assess its efficiency. To this end, we measure the number of necessary manual steps and compare them to the steps that are performed automatically as well as their computation time.

Manual Effort. Our RBT technique for software variants requires the user to manually assign *feature impact values* for each non-abstract feature in the SPL as described in Chapter 5.2.1. This value depends only on the number of non-abstract features in the SPL. For BCS, we had to manually assign 27 feature impact values based on expert knowledge. In addition, 32 feature mappings for elements of the core variant have been manually defined. We reduced this value as we only use mandatory features in the core variant. These manual steps are only performed once for the SPL and require expert knowledge. Our RBT approach for software variants does not require a manual assessment of failure probabilities for features, which is a clear contrast to the risk-based SPL technique by Hartmann et al. [HvdLB14].

Automatic Computation. Based on the core variant mappings and the feature impact values for the SPL, our RBT approach for software variants is able to perform automatic risk-based testing for an arbitrary number of product variants of the SPL. For the 17 product variants analyzed for BCS, our RBT technique performs a total of 1627 feature mappings automatically. It uses the applied deltas and their application conditions to derive feature mappings automatically. In addition, a total of 181 component failure impact values and, analogously, 181 component failure probability values are computed. This requires a total time of 356 ms, measured on a Intel Core i7-6600U CPU. This shows that our RBT technique for software variants is efficient, especially when compared to manual assignments. Performing several hundreds of risk assessments or risk computations manually is not feasible. Thus, we propose an efficient RBT approach for SPLs.

Exploiting SPL Characteristics. Our test case prioritization technique is tailored for SPLs. Thus, we assess if it actually scales for SPLs, i.e., if changes of product variants have an impact on the risk computation for changing PUT. Figure 5.6 shows the incremental changes of the failure impact (*impact*), *failure probability* (*fprob*) and resulting risk value (*risk*) of one example component, the automatic power window (*AutoPW*) in BCS. It shows the changes of these values for the product variants in the order they occurred in testing. The graph indicates that our RBT technique exploits changes of previously tested product variants compared to the current PUT to adapt the computed values. For example, the failure probability is descending from the fourth product variant under test until the last. These changes show that our assumption about an increasing number of reoccurring

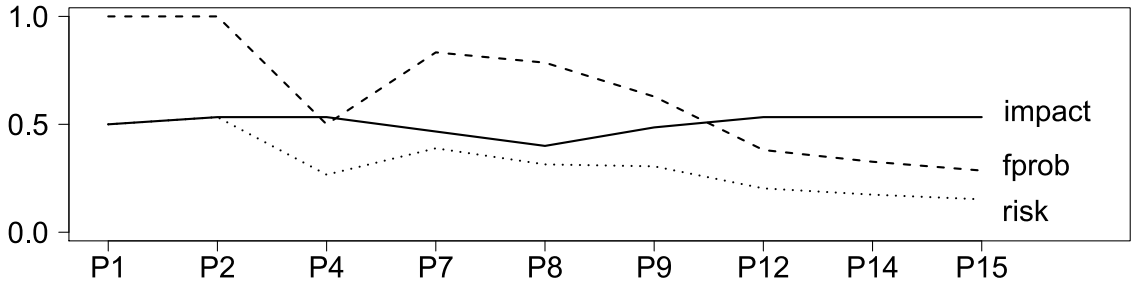

Figure 5.6: Incremental Risk Value Assessment for AutoPW (cf. [LBL+17])

interfaces in SPLs holds. In other words, we are able to exploit the reuse potential in SPLs to adapt our computations. In contrast to the failure probability, the impact value for *AutoPW* is very stable, which is due to the fact that impact values for features are stable and changes to this component are not drastic. However, the risk value is descending with a growing number of product variants. This reduces the test effort for later tested product variants and supports a test case prioritization of large SPL product spaces.

Consequently, we argue that our RBT technique scales for SPLs and is efficient. This validates our second research question. In comparison, traditional risk-based approaches require higher amount of effort, as they do not exploit or analyze changes between product variants. Our automatic analyses improve the risk-based test case prioritization efficiency. Especially our ability to avoid manual failure probability assignments increases the efficiency compared to traditional RBT techniques.

5.3.4 Threats to Validity

Internal Threats to Validity. The test artifacts influence the quality of our assessment. Thus, we use an established case study, which has been shown in the past to be of high quality [LLLS12, LLL+14, LLL+15]. The failure seeding influences the APFD values. It represents mutated failures, which are not necessarily realistic. Thus, we performed two different seeding experiments. We simulate failures in changes, which we assume to be the prime source of failures. While we focus on first order mutants, i.e., simple and few failures [DLS78], higher order mutants which are more complex might lead to other results and are worth to be investigated [JH08]. To avoid statistical outliers, we repeated our experiments 100 times. For BCS, we manually assigned feature impact values. These values heavily influence the approach and are of importance for the computation. Thus, our expert knowledge helped us to assign feature impact values and feature mappings for the core [LLLS12].

External Threats to Validity. As we apply our risk-based test case prioritization approach for software variants to one case study with seeded failure data, our results are not generalizable for each type and domain of SPLs. However, our experiments show the general applicability, effectiveness and efficiency of our RBT technique for software variants and lay the foundation for future research. The presented technique is based on heuristics, which we use to improve the efficiency and effectiveness of our RBT technique for software variants compared to traditional risk-based approaches. These assumptions might not hold for every domain of SPLs and are subject to change under certain circumstances, e.g., different types of software, product complexity or available data. However, our metrics are not designed to be rigid, but can be adapted to represent different assumptions, e.g., by adapting the analyzed types of test models or normalizations.

5.4 Related Work

Risk-based testing (RBT) has been widely used in single-software systems [ELR+14, FS14]. However, to our knowledge there is only one risk-based approach for SPLs proposed by Hartmann et al. [HvdLB14]. We present related work in RBT which uses similar approaches and artifacts as our RBT technique. An overview of related work is shown in Table 5.1.

RBT for SPL Testing. In terms of risk-based SPL testing, only one similar contribution is known to the author at the time of writing this thesis. **Hartmann et al.** [HvdLB14] present to our knowledge the only risk-based engineering approach specifically designed for SPLs. As explained earlier, they use quantified feature models to assign failure impact and failure probability values to each feature in the SPL. This can guide the SPL engineering, e.g., focusing on risky product variants first. In contrast to our RBT approach, they focus solely on product-level, i.e., they do not prioritize test cases for product variants. Furthermore, their require manual assignments of failure probabilities and do not dynamically compute risk values according to already tested product variants. In addition, they do not further specify how to use the risk-based knowledge in the context of SPL testing.

RBT for Single-Software Test Case Selection. Several risk-based test case selection techniques for single-software systems exist. **Adorf et al.** [AFVB15] present a risk-based test case selection technique. They use Bayesian statistical models to compute the *risk decrement*, i.e., whether a QA-task is executed or not. The authors use expert knowledge do derive the necessary input data for their Bayesian model. Their approach is neither model-based nor is it designed for variant-rich systems such as SPLs. **Chen et al.** [CPS02] propose a risk-based approach for test case selection. They trace paths within activity diagrams. These paths resemble test cases. Assigned to the paths are cost and severity values. They compute the

Table 5.1: Categorization of Related Work for Risk-Based Test Case Prioritization for SPLs

	Technique			Risk Computation		Input Artifacts		
Related Work	**Selection**	**Prioritization**	**SPL**	**Automatic**	**Manual**	**MBT**	**Code**	**Other**
RBT for SPL Testing								
Hartmann et al. [HvdLB14]			X		X			X
RBT for Single-Software Test Case Selection								
Adorf et al. [AFVB15]	X			(X)	X	(X)		X
Chen et al. [CPS02]	X				X	X		X
Felderer et al. [FHBM12]	(X)			X	X	X		(X)
RBT for Single-Software Test Case Prioritization								
Bai et al. [BKY12]	X	X		X	(X)			X
Hettiarachchi et al. [HDC14]		X			X			X
Hettiarachchi et al. [HDC16]		X		X			X	X
Kloos et al. [KHE11]	(X)	X		(X)		X		
Stallbaum et al. [SMP08]	(X)	X			X	X		
Yoon et al. [MYC12]		X			X			X
Our RBT Approach	(X)	X	X	X	X	X		

SPL = Technique applicable to SPLs, **MBT** = Model-based approach, **Code** = Technique requires code information

severity probability of a test cases based on the number of uncovered failures and their severity. In addition, they use a cost model for test cases, based on the cost of requirements. Test cases are executed according to their risk exposure. Compared to our risk-based approach, they do not analyze changes between different product variants of their models to make regression testing assumptions for different product variants. **Felderer et al.** [FHBM12] integrate manual and model-based risk assessments. They perform static analysis for automatic risk assessments, where failure probability and failure impact are derived by metrics. They also include time metrics, e.g., by taking different versions into account. Based on manual and automatic approaches, the authors are able to compute risk values for system artifacts to support testing. In contrast to this thesis, they do not focus on SPLs nor do they consider delta knowledge available for their artifacts.

RBT for Single-Software Test Case Prioritization. We present related risk-based test case prioritization techniques for single-software systems in the following. **Bai et al.** [BKY12] introduce a risk-based test case selection and prioritization technique. Their technique is tailored for evaluation of semantic web services. Semantic errors are more difficult to detect than syntactic errors. They extract failure importances and failure probabilities from ontology, service and composite service data. This provides an objective risk assessment. The extracted data is defined as ontology. Resulting, they are able to select and prioritize test cases. In contrast to our RBT technique for software variants, they require ontology information for their risk assessment and do not analyze risks for product variants in SPLs. **Hettiarachchi et al.** [HDC14] present a test case prioritization technique based on requirements and risk information. Based on expert knowledge, they perform a thoroughly analysis of the requirements and assign priority values to test cases according to the covered requirements. In contrast to this work, these analysis steps are performed manually and are not designed for SPLs or incremental testing. In following work, **Hettiarachchi et al.** [HDC16] present a risk-based test case prioritization approach. They derive a *requirements modification status* and *potential security threats* for requirements. They use fuzzy expert systems [Kan91], i.e., they automatically assess risk for requirements based on different indicators, such as requirements complexity. They use the source code and natural language requirement descriptions to calculate the risk indicators. The results of their technique allow them to prioritize requirements and test cases. Their results indicate that their approach finds failures early. In contrast to our work, they rely on code analysis and do not analyze SPLs. **Kloos et al.** [KHE11] introduce a model-based test case generation technique, which also supports the prioritization of test cases. They use fault state machines to derive test cases based on a fault tree analysis. This analysis reveals which events lead to which risks. Thus, they are able to prioritize the derived test cases according to their coverage of the system. The authors focus

on safety-critical systems. Their work is not designed for variant-rich systems and requires fault trees for the system under test. **Stallbaum et al.** [SMP08] introduce a test case generation approach based on annotated UML activity diagrams. Their test cases represent testing scenarios in the system, which abstract from specific test data. In addition, they prioritize the generated test cases based on risk values. Risk values are annotated to the test models after the risk assessment phase, which they do not further restrict or specify. In contrast to this thesis, they do not prioritize test cases for SPLs, nor do they specify an approach to generate risk values automatically for product variants. **Yoon et al.** [MYC12] propose a test case prioritization approach based on the analysis of requirements and risk. They prioritize new test cases instead of reusable ones. The required risk values are provided by test experts. The authors propose to use the *analytic hierarchy process* [Saa08] to support the decision making. Their approach requires a high manual effort and, thus, does not scale for SPLs.

5.5 Chapter Summary and Future Work

Summary. Risk-based testing is a popular approach in the quality assurance of single-software systems [ELR+14, FS14]. However, currently only one related contribution has been proposed to introduce risk-based software engineering for SPLs [HvdLB14]. This existing technique focuses on product-level assessments of risk values, which are manually defined. In contrast, we propose a risk-based approach for SPLs, which enables a test case prioritization for each product variant under test. Our manual effort is limited to feature impact value assignments and feature mappings for test model elements of one (core) product variant. Based on this, we are able to automatically compute feature impact and feature probability values for test model elements in the current PUT. Our analysis is based on knowledge about changes between product variants and their corresponding test models. The higher the risk value of a test model vertex and its neighborhood is, the higher is its importance for testing. Based on this knowledge, we prioritize test cases which cover system parts which propose a high risk. This greatly reduces the effort of risk-based testing compared to single-software RBT approaches. Especially, as manual effort is independent of the number of product variants under test. We use the BCS case study to evaluate our RBT approach. The evaluation results indicate that our RBT technique for software variants is effective as it finds important failures early in an integration testing scenario.

Future Work. To generalize our findings, additional case studies have to be performed. Especially case studies, which provide realistic failure data are required to increase the confidence in our results. In addition, we are interested in the possibility to use the risk-based approach in the context of search-based testing. For example,

risk-based testing can be used as objective in a multi-objective test case prioritization concept. Several objectives could be optimized at once to improve the failure finding rate, e.g., by using genetic algorithms (cf. Chapter 6.1) [McM11, HJZ15]. Other, additional objectives can be derived from our SPL framework. Thus, additional information can be integrated in our test case prioritization. Further potential future work is the derivation of guidelines and best practices for the definition of feature impact values. On feature model level, feature impact values can influence each other. For example, parent features could influence the feature impact values of their children. Consequently, sufficient rules to automatically adapt feature impact values are worth to be investigated.

Part III

Black-Box Testing of Software Versions

6 Multi-Objective Regression Test Case Selection for Software Versions

The content of this chapter is largely based on work published in [LFN+17].

Contribution

We present a multi-objective black-box regression test case selection technique for software versions. It is applicable to system-level testing, where no source code access is available. Instead, we use a wide variety of available black-box meta-data, such as failure history. We apply genetic algorithms to compute Pareto-optimal test sets. Results indicate that our test case selection approach outperforms a random test case selection and retest-all.

In this chapter, we present our novel multi-objective regression test case selection technique for regression testing of black-box software versions. Different techniques have been proposed to cope with the complexity of regression testing and reduce the testing effort in repetitive testing of different software versions [YH07b]. Test case selection has been introduced to select a certain subset of test cases as representatives, which have a high likelihood to reveal faults [RH94]. In recent years, test case selection has primarily been investigated for white-box testing [ERS10]. This restricts the applicability of such techniques in system-testing as source code is not always available, e.g., if components are developed by different suppliers or part of closed-source libraries.

Out novel regression test case selection is based upon the meta-data described in Chapter 3.1.2, i.e., requirements, test cases and failures described in natural language accumulated with meta-data. We utilize genetic algorithms to optimize the regression test case selection. Thus, these types of algorithms are explained first in the context of search-based software testing. Next, we present the regression test case selection technique for which we define seven objectives to be optimized. Our black-box objectives are combinable and allow for a very flexible test case selection applicable to different software domains. We use two different subject systems to evaluate the efficiency and effectiveness our regression test case selection technique. In addition, we compare the regression quality of our test case selection approach to a random test case selection and the retest-all regression testing approach. We present related work to show the current state of the art and the novelty of our regression test case selection technique compared to similar techniques.

6.1 Search-based Testing and Genetic Algorithms

Search-Based Testing. To improve software testing, *search-based software testing* (SBST) techniques have emerged and gained a lot of interest in recent years [McM11, HJZ15]. The commonality that connects all SBST techniques is that they are based upon meta-heuristic optimization search techniques such as genetic algorithms (GA) [Hol92, Deb01, McM11]. These techniques support the optimization of several objectives at once and, thus, lead to a *multi-objective* optimization [KCS06]. Due to the generic nature of SBST techniques, they have been applied to many different applications [SP94]. This thesis focuses on the application of SBST to solve the test case selection problem [RH94, YH07a] (cf. Definition 2.1 in Chapter 2.2.1). In particular, we apply a GA to solve black-box test case selection as multi-objective optimization problem [Hol92] without access to source code.

Introduction of GA. GAs adapt concepts of nature's evolution to find solutions to optimization problems whose best solutions are unknown [Dav87, LKM+99]. This allows GAs to overcome local optima due to different evolutionary concepts, e.g., mutation. A GA usually consists of the five main phased depicted in Figure 6.1. It begins with the *computation of the initial population*, followed by the GA cycle. While the first phase is executed once, the other four phases are repeated in an iterative cycle. In particular, the *fitness computation*, *selection*, *crossover* and *mutation* phases are part of the GA cycle. We explain the five different phases of GA in the following. These phases are later applied to find test sets according to the black-box test case selection objectives introduced in this thesis.

Compute Initial Population. GAs operate on a set of *individuals*, which represent the current solution candidates to the problem to be solved [Mit96]. The set of current individuals is referred to as the *population* of the algorithm. Similar to nature, each individual consists of a set of *genes*, which can be manipulated in certain ways to generate individuals which are better suited to solve the optimization problem. Each phase of the iterative algorithm operates on the current population and applies certain operators, which influence the output of the particular phase and, thus, the algorithm [LKM+99, Sar07].

To initiate a GA, an initial population has to be defined. Different ways exist to compute the initial population [DH07]. The main challenge in this phase is to find a good population size [PS06]. Influencing factors are the optimization problem difficulty, search space, number of individuals and population diversity [DH07]. Similar to nature, GAs start with an initial population that does not necessarily consist of adequate solution candidates. Instead, in the context of this thesis, the initial population is created in a random fashion. A common default size is 100 individuals, which is known to deliver good results for different optimization problems [RFP13]. The randomly selected individuals are used as offspring for future individuals.

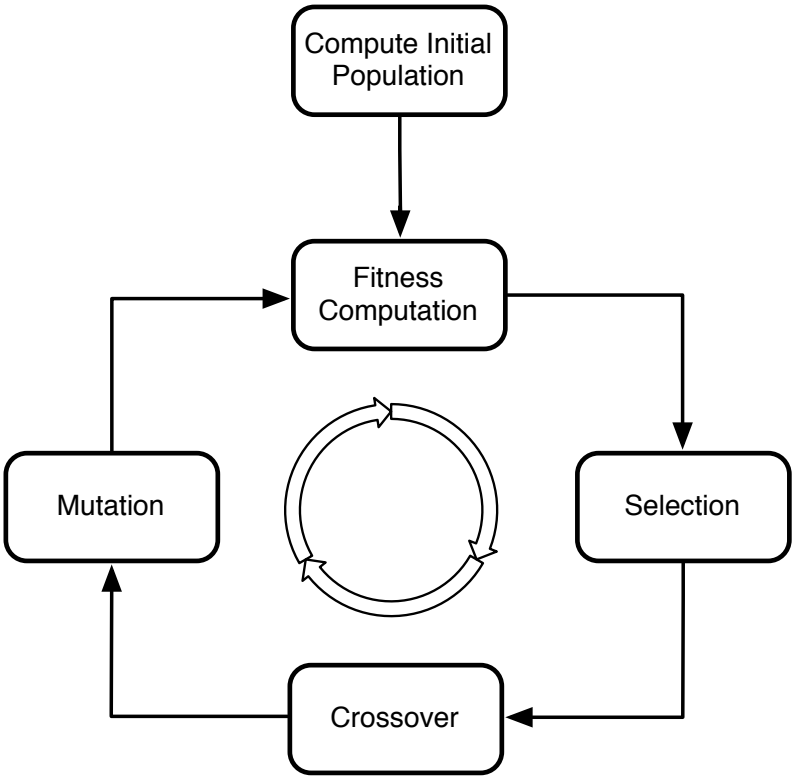

Figure 6.1: Genetic Algorithm Phases

Compared to the other four phases, the initial population phase is executed once, as the initial population is used as offspring for the following phases, which create new individuals to find the best solution candidates. However, as adequate initial populations have shown to influence the overall result, this step has a large impact on the subsequent phases [BGK04].

Fitness Computation. It has to be known how good a *solution candidate* (i.e., an individual) solves the investigated problem to find a proper solution to an optimization problem. In GAs, this is done by applying problem-specific *fitness functions*, which are formal representations of problem statements [WH06]. Designing suitable fitness functions is the key task to successfully solve a given optimization problem using a GA. Finding and defining optimization objectives is probably the most difficult task, as fitness function design requires a deep understanding of the problem [WT11]. More complex problems can be split into subproblems, each defining a separate objective to be optimized. If more than one objective is solved at once, the GA performs a *multi-objective optimization* [Deb01, KCS06]. As objectives might contradict each other, finding a multi-objective solution is a difficult task for which GAs have shown a lot of potential [SA13].

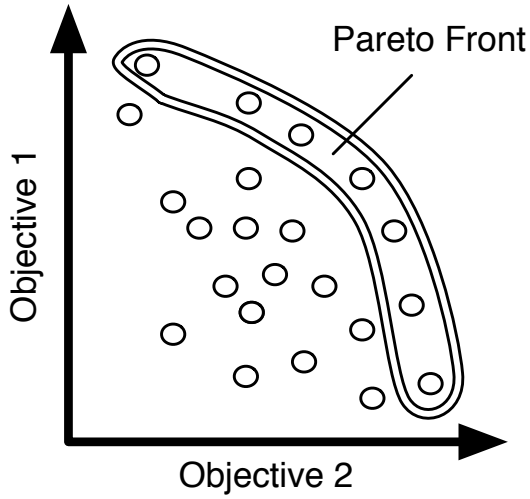

Figure 6.2: Sample Pareto Front

Selection. It is essential to find good solution candidates among the current population's individuals. As we are only interested in the best solution candidates, it is helpful to *select* a subset of candidates according to their fitness as offspring for the next generation, i.e., as population for the next GA iteration [BT96]. This *survival of the fittest* approach resembles the evolution process in nature. For multi-objective problems, each individual receives a separate fitness value for each objective. The fitness of an individual consists of all its fitness values in combination. Finding the best individuals for multi-objective problems requires the maximization of all objectives at once. These solutions are said to be *Pareto-optimal* and, thus, build the *Pareto front* (cf. Figure 6.2). Pareto-optimal (non-dominated) solutions share the ability to solve the optimization problems better than all dominated solution candidates. However, several Pareto-optimal solutions exist which differ in their fitness for the individual objectives [Cen77, DPAM02], i.e., they solve the different objectives to a different degree. Pareto-optimal solutions have in common that one objective cannot be further optimized without decreasing at least one other objective [Coe00]. Consequently, no Pareto-optimal solution is better than another w.r.t. all objectives.

In addition to finding the Pareto front, multi-objective algorithms aim to keep the population diverse to avoid *genetic drift*, i.e., a convergence of the solutions in one niche [Coe00]. Thus, the population density in the solution space is considered as additional criterion for the selection of new individuals.

Example 6.1: Computing the Pareto Front

Assume that a GA has finished its computation of a two-objective optimization problem. The resulting population is shown in Figure 6.2. Assuming that both objectives have to be maximized, the two-dimensional coordinate system shows that seven individuals (i.e., dots) build the Pareto front, not being dominated by any other solution candidate. While other Pareto solutions might have been possible, the algorithm tries to achieve a uniform distribution of results in the front.

Different outcomes of the selection phase are possible. First, some individuals might outclass a predefined fitness-based stop criterion, i.e., they provide a sufficiently good solution to the problem as their fitness surpasses a certain a priori defined threshold value. This stops the GA execution. Within the final population, a ranking of individuals is computed and the Pareto-optimal solution candidates are returned [KCS06]. Due to its nature, the Pareto front contains more than one solution. Second, a stop criterion has either not been defined or the specific value is not reached. In this case, the GA continues using the best individuals for the next phases in an iterative fashion. The selection depends on the chosen operators, but one popular approach is to select a certain number (e.g., 50%) of the best individuals based on their achieved fitness [BT96, CRK10]. Another popular way to find the best candidates is to use *tournament-selections* [MG95]. In this case, s random individuals compete against each other w.r.t. their fitness to solve the objective(s). The best candidate is taken into the new offspring set, the others are discarded or have to compete in another tournament.

If the final selection is performed after the GA has finished, the individuals of the Pareto front are returned as solution set. This means that only those solutions are returned, which are not dominated by other solutions, i.e., which are Pareto optimal.

Crossover. After a set of (best) individuals is selected, the GA creates a new set of *offspring* individuals. These new individuals are integrated into the current population along with the previously selected best individuals. The new offspring is created using *crossover* operators [Mit96]. They allow to combine existing individuals (i.e., *parents*) into new, different individuals (i.e., *children*). Different crossover operators have been defined for different problem domains [PC95, Mit96, MM13]. In most cases, two parents form two children. In case that 50% of individuals form the parents of an original population, creating exactly two children keeps the population size stable. This avoid an increase in computational effort.

One example for crossover operators are *n-point crossover* techniques, the simplest being a 1-point crossover. These operators cut the parents at n different points between their genes. Children inherit different parts of genes from both parents.

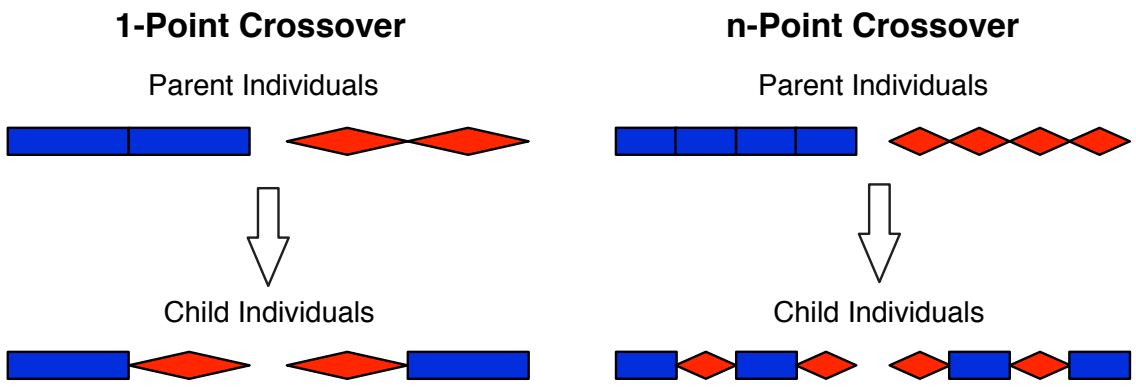

Figure 6.3: Crossover Operator Examples

Example 6.2: Crossover Operators

Figure 6.3 shows an example for two different crossover operators on two individuals, marked in blue and red. The example shows that the crossover operator splits both individuals n times to generate two child individuals. On the left hand side of Figure 6.3, only one split is performed. In this case, both children contain two original genes stemming from one parent each. On the right hand side, three splits are performed, representing an *n-point crossover* for $n = 3$. In our example, both techniques generate two children to keep the population size stable. Both, parents and children are part of the population.

Mutation. After new offspring individuals have been generated using crossover operators, the GA cycle could start from the beginning. However, this would reduce the number of possible solutions in the population as new offspring could only contain parts of already existing parents derived by crossover. This reduces the solution space, potentially keeping the GA at a local optimum. To avoid this inbreeding of individuals and introduce new possible solutions to the population, an additional *mutation* step is performed in the GA cycle [Mit96]. In this phase, certain genes of existing individuals are mutated, i.e., randomly changed. Thus, similar to the other phases of GA, different mutation operators have been introduced in the past and are applicable depending on the problem space and domain [HS11], e.g., they may add genes (*insertion mutation*), reorder them (*reciprocal mutation*), or switch one gene with other genes from the pool (*uniform mutation*). In contrast to crossover, no new individuals are added but existing ones are mutated. Consequently, the applied mutation operation depends on the optimization problem to be solved.

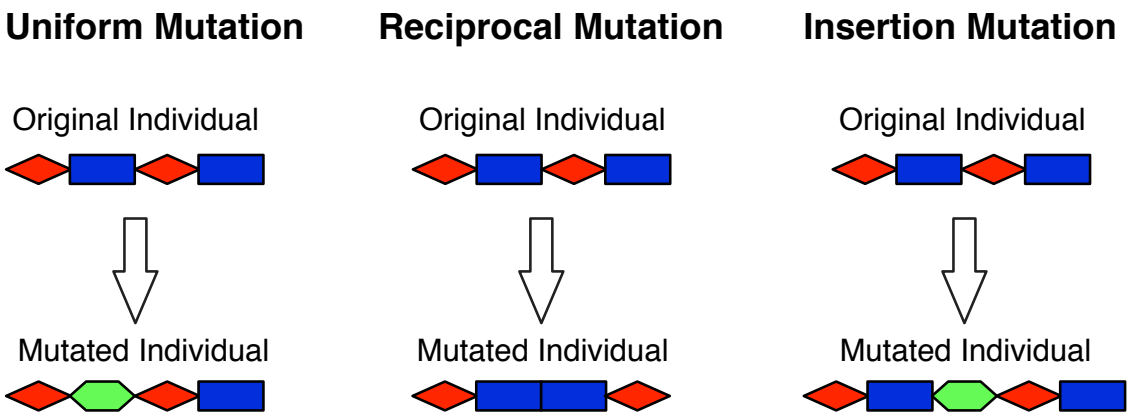

Figure 6.4: Mutation Operator Examples

Example 6.3: Mutation Operators

Figure 6.4 shows examples for potential mutation operators based on the individuals retrieved by the sample n-point crossover in Figure 6.3. The example shows how an individual is mutated in three different ways separately. First, one blue gene is replaced by a green gene. This gene has not been present before and shows the potential of introducing new information into the population to avoid inbreeding while keeping the size of individuals constant. The second example mutation switches the positions of two genes. Such a mutation is important when the order of genes is relevant for an individuals' fitness, i.e., in prioritization tasks. The third mutation extends the number of genes using insertion. This is shown on the right hand side of Figure 6.4, where a green gene is added. Similar to uniform mutation, this adds new information into the gene pool, but can also change the size of individuals, e.g., in selection tasks.

The *mutation rate* is another aspect that influences the mutation phase [OHB99, LLH03]. An excessive mutation of individuals is not desired as it contradicts the search-based nature of GAs and leads to a random search. To curb the mutation, the mutation rate might be that in average only one gene per individual is mutated in each cycle, or that only a certain number of individuals partakes in the mutation step. This keeps a large aspect of the original individuals and supports the crossover of individuals.

Once the individuals have been mutated one iteration of the GA cycle is complete and the next iteration is initiated, starting with the fitness evaluation of individuals. The GA cycle is repeated until a stop criterion of the GA is reached.

Stopping Criteria. As GAs are performed in an iterative fashion, the number of performed iterations influences the quality of the results [Kim10]. The faster the algorithm converges to the optimal solution, the more successful it is. While the search continues, new and better results might be revealed by the algorithm. However, searching may require a lot of computational resources due to a high number of performed GA iterations. Hence, it is important to define when to stop searching, while still obtaining good results [Kim10].

Different stopping criteria can be defined to determine when to finish the GA execution and avoid infinite loops [GM00, SCPB04]. A first basic idea is to restrict the number of iterations making sure that the GA stops after a maximum number of cycles. This might not lead to the best solution, but enables a better scheduling of the process. Another simple possibility is to restrict *computation time*. Here, a definitive end date can be set to make sure that a solution is present at a defined point in time. Again, it does not ensure a high quality solution. This can be circumvented using a *fitness-based* stop criterion. In this case, the GA stops after one or more solutions have reached a certain fitness as defined by the optimization objective(s). For multi-objective algorithms, the fitness has to be specific for at least one objective or several objectives at once. While a fitness-based criterion ensures that a certain threshold of quality is delivered, it has two major shortcomings besides the difficulty of formulating it for multi-objective optimization. On the one hand, it does not enforce that the algorithm ever reaches this goal, i.e., there might be no such solution and, thus, the GA might never stop. On the other hand, if a solution is reached which qualifies to the stopping criterion the GA finishes, even though it might would have revealed an even better solution in an reasonable amount of time. Thus, a combination of stopping criteria can help to avoid their shortcomings when applied in isolation.

Example 6.4: Combining Stopping Criteria

Assume that a GA is performed which shall provide a reasonable solution for a given search-based problem. As upper bound, a number of 5,000 iterations is set, as the time it takes to compute one iteration is known due to previous GA executions. In addition, a fitness-based stop criterion is applied, i.e., for a given metric a fitness of 80% is desired. If the algorithm finds a solution of this quality or better, it stops and returns the set of Pareto optimal solutions. If no such solution exists, the algorithm converges to a good solution in a maximum of 5,000 iterations preventing an endless execution.

6.2 Multi-Objective Regression Test Case Selection Approach

Test Case Selection. As software is updated regularly, regression testing of different software versions is an important task. Test case selection has been introduced to cope with large sets of regression test cases [RH94]. Based on a selection criterion, a subset of test cases is identified. The contained test cases are used as representatives of the available regression test cases (cf. Chapter 2.2.1). The quality of a regression test case selection depends on the selection objective, which is derived from changes between versions [RH97]. A variety of techniques has been proposed to realize regression test case selection, most of which are based on source code [ERS10]. However, especially in system testing or component-based software, source code is not always available.

In this thesis, we introduce a novel black-box regression test case selection approach based on the application of a multi-objective genetic algorithm [LFN+17]. The main workflow of our regression test case selection approach is depicted in Figure 6.5. We distinguish three main phases: *Data Preparation*, *Data Selection* and *Test Case Selection and Execution*. We explain the details of these three phases in the following.

6.2.1 Data Preparation

Before our test case selection technique can be applied, the available data has to be analyzed. Based on the black-box data artifacts gathered in software testing, which we described in Chapter 3.1.2, we introduce the input data for our regression test case selection technique and define objective functions to be optimized using a GA. Based on the available input data, we introduce the mathematical descriptions of seven black-box objectives, which we defined for black-box regression test case selection of software versions [LFN+17].

Data Extraction and Preparation

Available Data. Due to our restriction to black-box data, we assume that the artifacts defined in Chapter 3.1.2 are available in system testing. We do not require all data to be available. Instead, our test case selection technique is able to handle different subsets of these artifacts. In addition to the availability of system-testing related data, i.e.,*covered requirements*, *last execution time*, *revealed failure*, *failure priority* and *average test execution costs*, we incorporate *risk-related* data. Typically, risk information is gathered early in software projects to prioritize aspects in testing based on the risk of requirements [FHBM12]. As described in Chapter 5.1, risk is

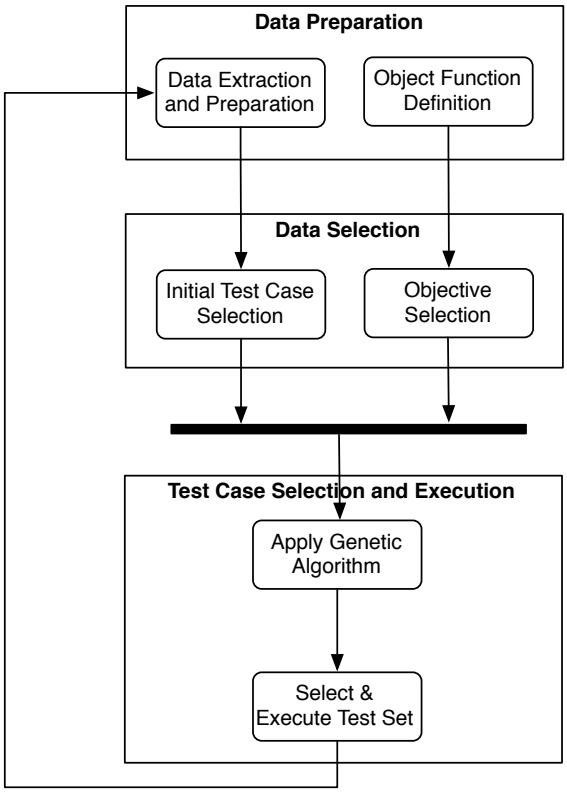

Figure 6.5: Multi-Objective Test Case Selection Technique (cf. [LFN+17])

something that can occur to parts of the system and might result in a loss or negative impact. In the testing context, the *risk* for an arbitrary risk item t is based upon the failure probability and the risk impact (cf. Definition 5.1 in Chapter 5.1) [Aml00].

Test Case Selection in Industry. Kasurinen et al. [KTS10] performed a qualitative study to assess how test cases are selected in industry. They distinguished between risk-based and design-based decisions made by experts. Their study revealed, that risk-based selections are favored when testing resources are limited and products are allowed to change. This supports the assumption of this thesis, that risk-based information supports regression test case selection in a positive manner and should be taken into account when performing selections. However, search-based testing has not yet focused on risk-based objectives [ELR+14], which is a novelty of our test case selection technique.

Risk Assessment. In contrast to our risk-based technique for software variants described in Chapter 5, we do not have model-based or SPL-based risk information available for versions. Instead, we assume that risk values are provided by system experts for the of system requirements $\mathcal{REQ} = \{req, \ldots, req_n\}$, where each require-

ment is linked to at least one system test case $tc \in \mathcal{TC}_{sys}$. The risk assignment task is typically performed manually at the beginning of a software project [FHPB14]. In addition, we assume that the risk impact is defined by the business value of a requirement and its corresponding functionality or feature. Hence, we define the *failure probability* and *business importance* in context of our test case selection technique for software versions as follows:

- *Failure Probability:* To assess the risk of a requirement, we define that result of the *failure probability* function $fp : \mathcal{REQ} \rightarrow \mathbb{N}$ is the likelihood that a requirement is not fulfilled. The failure probability is defined on a fixed nominal scale. In this thesis, we consider different scales, e.g., 1 to 5. If non-numerical values are used, they have to be transformed into a discrete mathematical representation. As intuition for failure probability, it can be assumed that the higher the likelihood to fail, the more important is a (re)test of the respective requirement.

- *Business Importance:* Besides the probability of failure, we also assume to know about the business importance $bimp : \mathcal{REQ} \rightarrow \mathbb{N}$ of requirements. This value represents the importance of a requirement in terms of a financial and economical standpoint. Similar to the failure probability, business importance is defined on a fixed nominal scale, which can differ depending on the domain or system under test (SUT). Test cases which cover highly relevant requirements should be retested with a higher probability than test cases for unimportant functionalities.

Example 6.5: Assessing Risk of Requirements

To assess the importance of requirements in risk-based testing, two-dimensional risk matrices are computed [XSWS09]. Figure 6.6 shows an example for a risk matrix. Given the two dimensions *business importance* and *failure probability*, we can arrange items in the matrix according to the risk values. In the example, four requirements A, B, C and D are shown. If we assume that the two dimensions represent two objectives to be maximized by a search-based algorithm, requirement D is of the highest priority followed by A and C. Hence, test cases related to these requirements should be favored compared to test cases which are related to requirement D.

Based on the available data in system testing including the described risk-related information, we define seven different black-box objective functions. They are used as optimization goals of the genetic algorithm applied to the test case selection problem for black-box testing.

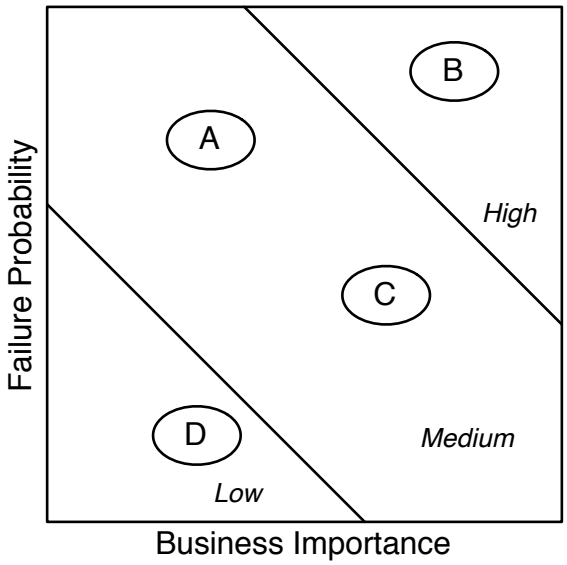

Figure 6.6: Sample Risk Matrix

Objective Function Definition

We define seven different objective functions to be optimized by a GA to realize the multi-objective regression test case selection [LFN+17]. In general, we aim to find a test set $TS \subseteq \mathcal{TC}$ which contains test cases which are likely to find a failure. Thus, our defined objectives represent different selection criteria for test cases. In the following, each objective is described in isolation as the objectives are freely combinable and do not depend on each other. This makes our regression test case selection approach very flexible. We do not normalize the objective functions, as we assume that we either want them to be the largest possible value (maximization) or the minimal possible value (minimization). The desired optimization goal is stated in the name of the objective, e.g., *Minimize Test Set Size*. The optimization is performed by a genetic algorithm which receives these functions as input for search-based optimization.

Minimize Test Set Size. Intuitively, test case selection aims to reduce the number of executed test cases in regression testing. Thus, solutions generated by the GA should contain less test cases than the overall number of applicable test cases. Therefore, we introduce the *Minimize Test Set Size* objective, with the goal to achieve a small number of test cases in a test set $TS \subseteq \mathcal{TC}_{sys}$. Naturally, if given enough time the minimization of the objective function $fit_{minsize} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ reduces the number of test cases in a test set TS up to a point where no test cases are left, i.e., $TS = \emptyset$. Hence, we argue that this particular objective should not be

used in isolation to avoid empty test sets. Consequently, the objective should be used as companion for other optimization objectives to achieve an actual selection of test cases, while fulfilling other objectives which require the presence of regression test cases, i.e., $TS \neq \emptyset$.

Definition 6.1: Minimize Test Set Objective

Let $\mathcal{TC}_{sys}$ be the set of system test cases.

We define the fitness function $\mathit{fit}_{minsize} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to *reduce the number of test cases* in a test set $TS \subseteq \mathcal{TC}_{sys}$ as:

$$\mathit{fit}_{minsize}(TS) = |TS|$$

Maximize Requirements Coverage. Testing against a definition of coverage criteria supports suitable testing strategies [ZHM97, AO08]. While code coverage can be applied in white-box testing, one prominent black-box coverage criterion is *requirements coverage* [WRHM06]. As requirements traceability is available in the context of this thesis (cf. Chapter 3.1.2), we assume to maximize the requirements coverage as objective to be optimized for regression test case selection. Hence, we formalize a fitness function $\mathit{fit}_{maxreq} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to be maximized. It sums the requirements Req_{tc} up, which are linked to each selected test case $tc \in TS \subseteq \mathcal{TC}_{sys}$. The higher the number of covered requirements, the better is the test set suited to achieve a good requirements coverage of the SUT. Using the *Maximize Requirements Coverage* objective in isolation leads to large test set sizes, as they achieve a high requirements coverage. When combined with the minimal test set objective, test cases gain a higher priority which cover more than one requirement at once. The objective combination resembles an intuitive way to increase coverage with less effort compared to a retest-all scenario.

Definition 6.2: Maximize Requirements Coverage Objective

Let $Req_{tc} \subseteq \mathcal{REQ}$ be the set of requirements linked to a system test case $tc \in \mathcal{TC}_{sys}$.

We define the fitness function $\mathit{fit}_{maxreq} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to maximize the number of *unique requirements* covered by a test set $TS \subseteq \mathcal{TC}_{sys}$ as:

$$\mathit{fit}_{maxreq}(TS) = \sum_{i=1}^{|TS|} |\{Req_{tc_i} \mid tc_i \in TS\}|$$

Maximize Failure Revealing History. Regression testing is concerned with the test of already tested parts of a system and examines their compliance to the specification [ERS10]. Another aspect of regression testing is failure retesting, i.e., verify that bug fixes for previously found failures have the desired effect. The function $F_{Rev} : \mathcal{TC}_{sys} \rightarrow \mathcal{P}(\mathcal{F})$ returns the revealed failures $fail \in \mathcal{F}$ for a test case $tc \in \mathcal{TC}$. We assume that traceability between test cases and their revealed failures is provided. Thus, we are able to ensure that test cases which previously revealed failures are selected with a higher probability than test cases without a positive failure history. Furthermore, test cases which have revealed more failures than others might be more valuable in finding new failures. Also, important failures are more important for retesting than those with low impact on the system. For each failure, a failure priority value is returned by the function $prio : \mathcal{F} \rightarrow \mathbb{N}$ as described in Chapter 3.1.2. Hence, we define a fitness function $fit_{failureHistory} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ which measures the number of failures revealed in the past by a test set $TS \subseteq \mathcal{TC}_{sys}$. To this end, we sum the priority values of all failures revealed by all test cases in a test set. A combination of this objective with the minimization objective reveals small test sets which have a high number of previously successful test cases. This supports failure retesting.

Definition 6.3: Maximize Revealed Failure Coverage Objective

Let $\mathcal{F}$ be the set of revealed failures, $F_{Rev} : \mathcal{TC}_{sys} \rightarrow \mathcal{P}(\mathcal{F})$ the function which returns the previously revealed failures by a system test case $tc \in \mathcal{TC}_{sys}$ and $prio : \mathcal{F} \rightarrow \mathbb{N}$ a function returning failure priority for a failure $fail \in \mathcal{F}$.

We define the fitness function $fit_{failureHistory} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to *maximize the retest of previously revealed failures* of a test set $TS \subseteq \mathcal{TC}_{sys}$ as follows:

$$fit_{failureHistory}(TS) = \sum_{i=1}^{|TS|} \sum_{j=1}^{|F_{Rev}(tc_i)|} \{prio(fail_j) \mid fail_j \in F_{Rev}(tc_i), tc_i \in TS\}$$

Minimize Execution Cost. In regression testing, the retest-all approach is not feasible if the number of test cases is larger than the available time or resources to execute them. To guide a test case selection, it is of interest to achieve a high test coverage in a certain time frame. Hence, the execution cost returned by the function $cost : \mathcal{TC}_{sys} \rightarrow \mathbb{N}$ (cf. Chapter 3.1.2) of test cases is of interest to select fast executed test cases. We use the costs to formulate the fitness function $fit_{mincost} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$, which summarizes the costs of all test cases within a test set $TS \subseteq \mathcal{TC}_{sys}$.

Definition 6.4: Minimize Execution Costs Objective

Let $cost : \mathcal{TC}_{sys} \rightarrow \mathbb{N}$ be the function which returns the cost of a system test case $tc \in \mathcal{TC}_{sys}$.

We define a fitness function $fit_{mincost} : \mathcal{PTC}_{sys}) \rightarrow \mathbb{N}$ to *minimize the cost of test executions* of a test set $TS \subseteq \mathcal{TC}_{sys}$ to be minimized as follows:

$$fit_{mincost}(TS) = \sum_{j=1}^{|TS|} \{cost(tc_j) \mid tc_j \in TS\}$$

Maximize Last Test Case Execution. Even though the effectiveness of test cases is unknown a priori, testers aim to execute every test case at least once in the product life-cycle. This ensures a full coverage of all requirements under the assumption that each requirement is linked to at least one test case. In addition, test cases which have not been executed for a long time might be able to produce new failure findings due to changes in the system. Hence, we introduce a fitness function $fit_{lastexec} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$, which computes the time since the test cases in a selected test set have been executed. The longer the last execution has been due, the higher the priority of the corresponding test case. To reduce the computational complexity to optimize this objective, we use a predefined nominal scale to represent the time frame as shown in Table 3.3 in Chapter 3.1.2. Formally, the function $lastExec : \mathcal{TC}_{sys} \rightarrow [1, 7] = \{x \in \mathbb{N} \mid 1 \leq x \leq 7\}$ returns the last execution date for each test case. According to our scale, the lowest priority is given to test cases executed in the last week as we assume that new software versions are not released more often than once a week. The second-highest priority is given to test cases which have not been executed for more than 90 days. The highest value is given to test cases, which have never been executed within a project's life-cycle. We assume that these test cases should receive a special treatment in order to achieve a high test coverage within a project's life-cycle by executing every test case at least once over the course of a project. This also ensures that each requirement is tested at least once, assuming that each requirement is linked to at least one test case. Combining this feature with the *Maximize Requirements Coverage* objective reduces the amount of redundancy between test cases while focusing on never executed test cases which are linked to many requirements.

Definition 6.5: Last Test Case Execution

Let $lastExec : \mathcal{TC}_{sys} \rightarrow [1,7] = \{x \in \mathbb{N} \mid 1 \leq x \leq 7\}$ be the function which returns the date of last execution for a system test case $tc \in \mathcal{TC}_{sys}$.

We define the fitness function $fit_{lastexec} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to *maximize the last test execution* for a test set $TS \subseteq \mathcal{TC}_{sys}$ as:

$$fit_{lastexec}(TS) = \sum_{j=1}^{|TS|} \{lastExec(tc_j) \mid tc_j \in TS\}$$

Maximize Failure-Probability of Requirements. Based on the notion of risk-based testing, we aim to maximize the coverage of requirements with a high failure-probability [FS14]. The function $fp : \mathcal{REQ} \rightarrow \mathbb{N}$ returns the predefined failure probability for a given requirement $req \in \mathcal{REQ}$ (cf. Chapter 6.2.1). This ensures that functionality with a high likelihood to fail is tested with a higher priority than other functionality. Thus, we introduce the fitness function $fit_{fprob} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to maximize the covered requirements' failure probabilities. The failure probability is defined by a fixed nominal scale, as described in Chapter 6.2.1. The *minimize test set size* or *minimize execution cost* objectives have to be added to perform a test case selection.

Definition 6.6: Maximize Failure Probability Objective

Let $\mathcal{TC}_{sys}$ be the set of system test cases, $Req_{tc} \subseteq \mathcal{REQ}$ the set of linked requirements for a test case tc and $fp : \mathcal{REQ} \rightarrow \mathbb{N}$ the function which returns the failure probability for a requirement $req \in \mathcal{REQ}$.

We define the fitness function $fit_{fprob} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to *maximize the overall failure probability* of linked requirements in a test set $TS \subseteq \mathcal{TC}_{sys}$ as follows:

$$fit_{fprob}(TS) = \sum_{j=1}^{|TS|} \sum_{k=1}^{|Req_{tc_j}|} \{fp(req_k) \mid req_k \in Req_{tc_j}\}$$

Maximize Business Relevance of Requirements. Similar to the maximization of failure probabilities, risk-based testing also focuses on the business relevance of requirements, i.e., the resulting impact of a failure of a particular requirement [Aml00]. In this thesis, we assume that a business relevance value can be assigned to each requirement, defined as $bimp : \mathcal{REQ} \rightarrow \mathbb{N}$. The business relevance might resemble the importance of the requirement to the stakeholders or to the over-

all functionality of the system. If the business relevance information is available, we are able to use an objective which aims to maximize this value for a set of test cases. Consequently, we formally introduce a fitness function $fit_{BR} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$, which aims to maximize the sum of business values of all linked requirements within a test set, leading to a high overall business importance of the test set.

Definition 6.7: Maximize Business Relevance Objective

Let $Req_{tc} \subseteq \mathcal{REQ}$ be the set of requirements linked to a system test case $tc \in \mathcal{TC}_{sys}$ and $bimp : \mathcal{REQ} \rightarrow \mathbb{N}$ the function to retrieve the business relevance of a requirement $req \in \mathcal{REQ}$.

We define the the fitness function $fit_{BR} : \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{N}$ to *maximize business relevance* for a test set $TS \subseteq \mathcal{TC}_{sys}$ as:

$$fit_{BR}(TS) = \sum_{j=1}^{|TS|} \sum_{k=1}^{|Req_{tc_j}|} \{bimp(req_k) \mid req_k \in Req_{tc_j}\}$$

Combining the *failure probability* and *business relevance*-related objectives yields a risk-based testing approach [Aml00]. A current survey in risk-based testing shows that this is the first time that risk-based objectives are used for regression testing in search-based testing of black-box software versions [ELR+14].

6.2.2 Data Selection

Based on the introduced seven objective functions $\mathcal{O} = \{o_1, \ldots, o_7\}$ (cf. Chapter 6.2.1), we are able to perform a black-box regression test case selection based on GAs. In order to perform a regression test case selection using the defined data and objectives, the user has to decide which system test cases are used as input test universe $\mathcal{TC}_{sys} = \{tc_1, \ldots, tc_n\}$ and which objectives $O_{selected} =\subseteq O$ are to be applied to achieve a test case selection for a set of system test cases. Of course, if more objectives are defined and applicable, these can be included as well, e.g., to also support white-box testing.

We extracted three main factors which influence the selection of the input test case universe:

- *Applicability of test cases:* A test case selection should only be performed on test cases which are applicable to the current version of the SUT. Certain features might not be ready for testing and, thus, corresponding test cases can not be executed and should not be considered as input. Older test cases might be obsolete due to feature changes within the project life-cycle.

- *Features under test:* Regression testing might focus on certain features of the system and, thus, only corresponding test cases should be considered as input for a selection. This is a software project specific criterion.
- *Number of available test cases*: If there exists a large number of system test cases applicable to the current software version, it might be useful to preselect a subset of test cases to reduce the computational overhead and exclude test cases of less importance. This requires expert knowledge.

Alongside the initial selection of test cases used as input for our test case selection approach, a set of objectives has to be chosen as well. Due to our multi-objective approach we are able to select i objectives out of all objectives $\mathcal{O}$ ($i \in \mathbb{N}, 1 \leq i \leq 7$) used as optimization criteria for test case selection. This allows for a very dynamic and flexible test case selection technique as objectives can be selected based on the availability of the respective data.

6.2.3 Test Case Selection and Execution

GA and Objective Selection. Based on defined objectives and available data, we execute our multi-objective regression test case selection for black-box software versions. We use a multi-objective genetic algorithm as more than one objective is optimized at once [KCS06]. Several genetic algorithms exist [SP94, Coe00]. While their underlying technology might differ, our test case selection approach is not necessarily restricted to a specific technique or genetic algorithms, as long as the chosen technique is suited to solve several optimization objectives at once. Using a multi-objective GA, we are able to apply a subset of objectives $O_{selected} \subseteq \mathcal{O}$ with $\mathcal{O} = \{o_1, \ldots, o_7\}$ being the set of all objectives defined in this thesis, i.e., in case of this thesis, $\mathcal{O}$ contains the seven defined objectives in Chapter 6.2.1. Applying the objectives using a GA for a set of test cases leads to a regression test case selection. In addition, a suitable stop criterion has to be selected for the GA execution. We described some basic techniques earlier (cf. Chapter 6.1). However, finding a sufficient stop criterion is non-trivial [SCPB04], which makes it hard to make suggestions. In any case, there should be sufficient number (>1,000) of iterations to ensure that a GA has enough time to find suitable solutions.

Terminology. In our test case selection approach, each *individual* of the population represents a test set. Individuals are represented as bit-vectors. Each bit corresponds to a distinct test case of the set of all system test cases $\mathcal{TC}_{sys}$, i.e., a gene represents a single test case. In particular, a gene with a value of 0 represents that the corresponding test case is not selected in the current test set, while a value of 1 indicates that the test case is present. This allows to easily add and remove test cases using mutation and crossover operators. While we do not restrict our

test case selection technique to certain operators, we restrict the types of allowed operations. Mutation and crossover operators shall not switch gene positions, which would result in a prioritization. Instead, operators change the value of genes, i.e., we use bitflip-mutations to represent adding and removing of test cases. Hence, each individual has the same length.

If all required GA parameters and operators are defined and configured, the GA is applied to select test cases based upon the chosen objectives. Coping with a range of objectives, algorithms and operators makes our test case selection approach very flexible. In particular, the *extensibility* of our test case selection approach has to be highlighted, as a GA basically builds a framework which can be extended using additional objectives, e.g., white-box-based objectives.

Selecting a Pareto Optimal Test Set. The result of a multi-objective GA is a set of Pareto optimal solutions [DPAM02, KCS06]. Let the function $optimize : \mathcal{P}(\mathcal{O}) \rightarrow \mathcal{P}(\mathcal{TC}_{sys})$ return the set of Pareto optimal solutions by applying the genetic algorithm for a set of objectives $O_{selected} \in \mathcal{O}$. In case of our test case selection approach, the result of applying a set of selected objectives $O_{selected}$ returns a set of Pareto-optimal test sets $TS_{Pareto} = \{TS_1, \ldots, TS_n\}$ such that $\{\forall TS \in TS_{Pareto} \mid TS \subseteq \mathcal{P}(\mathcal{TC}_{sys}) \wedge TS \in optimize(O_{selected})\}$ holds. Each of these test sets is different, but fulfills the optimization goals equally well, being not Pareto dominated by any other test set, i.e., no objective can be improved without deteriorating another objective. The number of Pareto-optimal test sets depends on the selected objectives $O_{selected}$ and input data, but does not exceed the population size used in the GA as each individual represents one (Pareto-optimal) solution. Usually, only a subset of the solutions is Pareto-optimal.

For test case selection, only one of the Pareto-optimal test sets is selected and executed. Hence, an expert has to decide which Pareto-optimal test set is used as representative. The selection is based on different criteria, the most important being the size of the test sets. Other criteria might be focused on the optimized objectives, e.g., when one objective is of higher priority to the project than others.

Example 6.6: Selecting a Pareto-optimal Test Set

Assume that the GA has generated a set of Pareto-optimal solutions explained in Example 6.1. For this example, let us assume that the following two objectives have been selected: *Minimize Test Set Size* and *Maximize Business Relevance*. Thus, each individual (i.e., dot) represents a test set. In Figure 6.2, the Pareto front contains seven test sets, which are equally well suited solution to the given optimization problem. However, for testing the tester might make the final decision of which test set to select based on the size of the test sets as it differs heavily between solution candidates.

Executing Test Cases. Once a Pareto-optimal test set is selected, the contained set of test cases $TS_{selected}$ is executed. Compared to test case prioritization, test case selection requires an execution of all test cases contained in the test set as all are of equal priority [YH07b]. We assume that test cases are executed manually as we focus on system-testing [Sne07]. Failures, which are captured in this phase are reported in a test management system and linked to the revealing test cases to enable traceability for future GA applications. This influences future selections, leading to new Pareto-optimal solutions. After all test cases have been executed, a new selection for upcoming versions can be performed.

6.3 Evaluation

In this section, we describe two research questions which guide the evaluation, the subject systems to which the approach has been applied to, our evaluation implementation and methodology and we present and explain the results we obtained. In addition, we present potential threats to validity.

6.3.1 Research Questions

To assess the feasibility and suitability of our test case selection technique, we formulate two research questions to be answered in our evaluation:

RQ1: *How* suitable *is our test case selection approach to solve the multi-objective black-box regression test case selection problem?* Test case selection aims to identify important test cases. In case of this thesis, we aim to select test cases which have a high likelihood to reveal new failures. Hence, we measure the effectiveness of our test case selection technique in terms of precision and recall. To assess the suitability of our test case selection, we compare the results to a random test case selection and the retest-all approach.

RQ2: *How* efficient *and* adaptable *is our regression test case selection technique, i.e., is it is applicable to incomplete artifacts?* We aim to present an efficient and flexible test case selection technique that is applicable to different black-box subject systems, objectives and, thus, available input data for testing. We measure how efficient the execution of our test case selection approach is. In addition, we execute combinations of objectives and evaluate if they lead to statistically significant results.

6.3.2 Subject Systems

We apply our test case selection approach for software versions to two different subject systems for evaluation, which are explained in the following.

Body Comfort System. The Body Comfort System (BCS) case study has been described in detail in Chapter 3.2. To test software versions, we apply our test case selection techniques to the artifacts described in Chapter 3.2.2 [LLLS12]. The system-level artifacts have been defined by students with background knowledge about BCS. For our evaluation, we manually assigned risk-values for the provided system requirements. A total of seven failures have been seeded for the system-level test cases, linked to one test case each (cf. Chapter 3.2). As BCS is a academic system, we did not execute the system for actual testing, but retrieved the failure information from previous years, where students implemented the system on actual hardware. BCS does not provide actual test execution times or information about last test executions.

Industrial Data. An industrial real-life case study has been provided by an partner in the automotive industry. It comprises more than 5,000 test cases, 5,000 requirements and 2,500 faults. All assets are provided in natural language and additional meta-data. Test cases are executed manually. The data is provided in the test management tool HP Quality Center (QC)[1]. Traceability between the different artifacts is not always provided. The data has been collected during a span of several years by different system experts and, thus, represents a real life representative for data encountered in complex software projects. For the evaluation of our multi-objective test case selection approach we manually selected a subset of 577 test cases. These test cases revealed a total of 34 failures in previous executions. The industrial data set does not contain any risk-related information for test cases.

6.3.3 Implementation

Prototype. Out test data is provided in HP QC. We have access to requirements, test cases and failures. We prototyped our test case selection technique to perform the evaluation. The prototype extracts data from QC and transforms it into a binary representation (cf. Chapter 6.2.3). Our prototype is based on the jMetal-framework [DN11]. This framework supports a wide variety of different genetic algorithms and operators. We selected the NSGA-II algorithm, a standard GA implementation to evaluate our test case selection technique [DPAM02]. In addition, we implemented the different GA phases based on certain operators provided by framework. These operators as well as their configuration used for evaluation are

[1]QC is part of HP Application Lifecycle Management, Website: `http://www8.hp.com/uk/en/software-solutions/application-lifecycle-management.html`, Date: April 11th 2017

Table 6.1: Configuration of applied NSGA-II Operators

GA Phase	Applied Operator	Configuration
Number of Iterations	-	Number = 5,000
Initial Population	Random Selection	Size = 100
Encoding	Binary	Length = $\|TC\|$
Selection	Binary Tournament	-
Crossover	HUX-Crossover	Chance = 0.9
Mutation	Bitflip-Mutation	Chance = $\frac{1}{\|TC\|}$

shown in Table 6.1. Both, operators and configurations of the algorithm have been chosen according to the given problem, i.e., test set selection. As explained in Chapter 6.2.3, crossover and mutation operators are limited to those applicable to the problem of regression test case selection, i.e., in case of this thesis genes are not switched, but their value is swapped between 0 and 1. We implemented operators which supported the optimization problem and have been available in jMetal.

Applied Operators. For data representation of test cases, we use bit vectors. A 0 represents a test cases that has not been selected and 1 means that the corresponding test case is selected. Thus, we can use bit-flip mutation operators to either select or remove a test case. Each individual has an average chance of one flip per mutation, i.e., one test case is added or removed on average.

We use the *Half Uniform Crossover Scheme* (HUX) to generate new offspring in the crossover phase [Mit96]. The offspring will probably adopt half of both parents' genes. However, the probability of inheriting a gene is computed anew for each gene of the parents. The distribution with which bits are selected from a parent is random, but with the chance of 50% for each gene to be from one particular parent.

We select the best individuals using a binary tournament selection implementation in jMetal, which is based on the binary tournament selection used for NSGA-II [DPAM02] (i.e., we use the *BinaryTournament2*[2] class provided in jMetal). In a binary tournament selection, two randomly chosen individuals compete against each other in terms of their fitness.

As we are interested in Pareto-optimal solutions, diversity of results is also a desired effect to avoid nearly identical solutions [TKK96]. To this end, the chosen binary tournament selection also applies a crowded-comparison procedure [DPAM02] to guarantee a high diversity among solutions. Therefore, the crowding-distance of

[2]Official jMetal API entry: `http://jmetal.sourceforge.net/javadoc/jmetal/operators/selection/BinaryTournament2.html`, date: April 11th 2017

Table 6.2: Confusion Matrix - Prediction vs. Outcome

	Predicted: Positive	**Predicted: Negative**
Actual: Positive	True Positive	False Negative
Actual: Negative	False Positive	True Negative

individuals is computed, which is a density measure of solutions in the neighborhood of the corresponding individuals [DPAM02]. The winner of each tournament is selected for the population of the next iteration of the GA. Thus, we select the 50% of the individuals for the following GA phases.

The population size on which the operators are applied is set to 100 individuals. This value has shown good results in the literature and has been identified as a standard value for GA [RFP13]. The number of iterations has been set to 5,000 to ensure that good Pareto optimal solutions are found. This is also the stop criterion.

6.3.4 Methodology

For each of the two subject systems, we assess the quality of our test case selection technique to answer the two research questions defined in Chapter 6.3.1.

Applied Objectives. We measure the effectiveness, efficiency and applicability of our test case selection approach using all possible objective combinations based on the seven objectives defined in Chapter 6.2.1. Of the available seven objectives, the *minimize test set size* objective has always been selected. This is done to realize an actual selection of test cases. Two other objectives have been applied to both subject systems: *Maximize requirements coverage* and *Maximize Failure History*. Both risk-based objectives *Maximize Failure Probability* and *Maximize Business Relevance* have only been applied to BCS as the required information is not available for our industry data. The *Minimize Execution Cost* and *Maximize Last Test Case Execution* objectives have been applied to industry data only as the information was not available for BCS due to artificially seeded failures. Therefore, we performed $2^4 - 1 = 15$ combinations of objectives for each system, each including the *Minimize Execution Cost* objective. As GAs are non-deterministic, we repeated our evaluation *ten times* to prevent statistical bias. All results presented in the following are the average values of these ten repetitions.

Quality Metrics. Different metrics are used to answer the effectiveness of our test case selection. To assess the quality of the results of our test case selection technique, we examine the test case selection according to the confusion matrix and derived metrics [Faw06]. Table 6.2 shows a simple confusion matrix.

In case of this thesis, a *positive* prediction corresponds to a test case which is selected, i.e., it is suspected that the test case reveals a failure when executed. In

contrast, predicting a *negative* value for a test case leads to not selecting the test case. While *true positives* and *true negatives* are the desired output of a classification technique (cf. Table 6.2), *false positives* are favored compared to *false negatives*. This is due the fact, that a false negative prediction indicates that a failure finding test case has not been selected, i.e., the corresponding failure is not revealed. False positives do not reveal a failure, but they increase the number of test cases to be executed by the tester. This reduces efficiency, but increases test coverage. Based on the confusion matrix, we assess the selection quality with the following metrics: *precision*, *recall* and *F-score* [GG05].

Precision. This thesis aims to identify test cases which have a high likelihood to reveal failures. *Precision* measures the ratio of failure revealing test cases out of the set $\mathcal{TC}_F \subseteq \mathcal{TC}$, which have been selected in a test set $TS \subseteq \mathcal{TC}$.

Definition 6.8: Precision metric

Let $\mathcal{TC}_F \subseteq \mathcal{TC}_{sys}$ the set of failure revealing system test cases.

We define the *precision* function $precision : \mathcal{P}(\mathcal{TC}) \times \mathcal{P}(\mathcal{TC}) \rightarrow \mathbb{R}$ for a test set $TS \subseteq \mathcal{TC}_{sys}$ based on ratio of failure revealing test cases as follows:

$$precision(TS, \mathcal{TC}_F) = \frac{|\mathcal{TC}_F \cap TS|}{|TS|}$$

Recall. *Recall* measures how many of the overall failure finding test cases $\mathcal{TC}_F \subseteq \mathcal{TC}_{sys}$ have been selected in a test set $TS \subseteq \mathcal{TC}_{sys}$. The more failure revealing test cases are selected, the higher is the recall value. This metric is of theoretical nature as it is only observable if all failures in the system are known a priori. In case of this thesis, failures are known for both subject systems in the evaluation and, thus, the metric is applicable.

Definition 6.9: Recall metric

Let $\mathcal{TC}_F \subseteq \mathcal{TC}_{sys}$ the set of failure revealing system test cases.

We define the *recall* function $recall : \mathcal{P}(\mathcal{TC}_{sys}) \times \mathcal{P}(\mathcal{TC}_{sys}) \rightarrow \mathbb{R}$ for a test set $TS \subseteq \mathcal{TC}_{sys}$ based on number of selected failure finding test cases as follows:

$$recall(TS, \mathcal{TC}_F) = \frac{|\mathcal{TC}_F \cap TS|}{|\mathcal{TC}_F|}$$

F-Score. To give an overview of precision and recall of a test set we compute the *F-score*. It aggregates both precision and recall. We use this metric to assess the statistical significance of different Pareto fronts and result distributions of our test case selection technique. The higher the F-score, the more failures have been found (due to high recall), using only a small subset of all test cases, i.e., selecting test cases which actually reveal failures (due to high precision). High recall and precision are the goals of a good test case selection [RH94, ERS10].

Definition 6.10: F-score metric

Let $\mathcal{TC}_F \subseteq \mathcal{TC}_{sys}$ be the set of failure revealing system test cases, $precision : \mathcal{P}(\mathcal{TC}) \times \mathcal{P}(\mathcal{TC}) \rightarrow \mathbb{R}$ the precision of a test set and $recall : \mathcal{P}(\mathcal{TC}) \times \mathcal{P}(\mathcal{TC}) \rightarrow \mathbb{R}$ the recall of a test set.

We define the *F-score* function $FScore : \mathcal{P}(\mathcal{TC}) \times \mathcal{P}(\mathcal{TC}) \rightarrow \mathbb{R}$ for a test set $TS \subseteq \mathcal{TC}_{sys}$ based on its precision and recall as follows:

$$FScore(TS, \mathcal{TC}_F) = 2 \cdot \frac{precision(TS, \mathcal{TC}_F) \cdot recall(TS, \mathcal{TC}_F)}{precision(TS, \mathcal{TC}_F) + recall(TS, \mathcal{TC}_F)}$$

Realization of Random Selection. We compare our test case selection against a random selection. To ensure a fair comparison, we create several random selections. In particular, for each Pareto-optimal test set derived by applying our test case selection technique, we create 100 random test sets. These randomly generated test sets are of the same size as the Pareto-optimal solutions for which they are generated. We measure the average precision, recall and F-score for these 100 random test sets to compare these results to our test case selection technique. We also compare our test case selection approach to retest-all, for which we assume to execute each test case for each subject system.

Example 6.7: Creating Random Test Sets

Assume two Pareto optimal test sets TS_1 and TS_2 generated by the multi-objective test case selection. Further assume that $|TS_1| = 23$ and $|TS_2| = 42$. To compare their quality to a random selection, we create 100 test sets containing 23 randomly selected test cases as comparison for TS_1 and 100 random test sets of size 42 for TS_2. Afterwards, we compute the average precision, recall and F-score for each group of 100 random test sets.

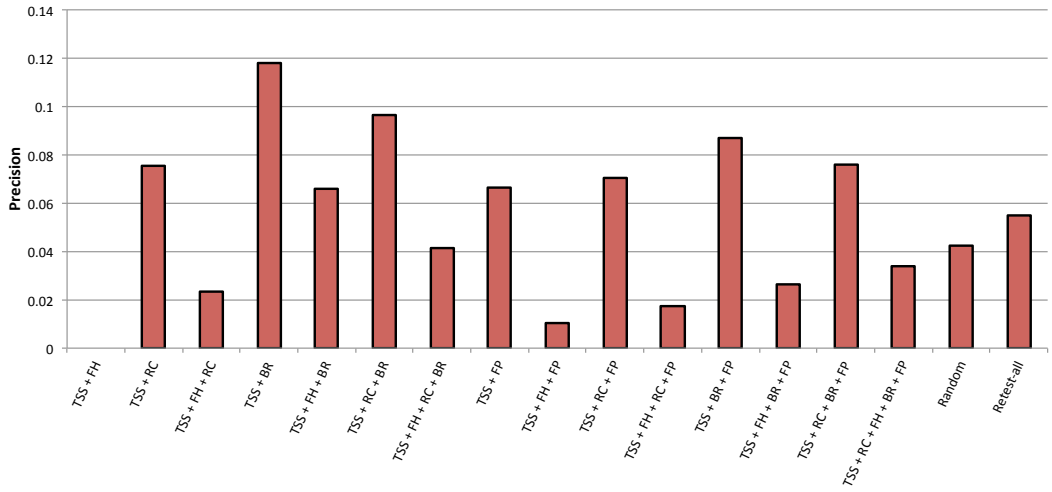

Figure 6.7: Aggregated Precision Values for BCS

6.3.5 Results and Discussion

We answer both research questions in the following, using results obtained by applying our test case selection technique to both subject systems.

> **RQ1:** *How* suitable *is our test case selection approach to solve the multi-objective black-box test case selection problem?*

To answer the first research question, we assess the effectiveness of our test case selection in terms of failure finding quality.

Precision Values for BCS. We measure precision of our selection technique for 15 different objective combinations for BCS. In total, seven failures have been seeded to be found, thus, about 5% of test cases reveal failures. A bar chart showing the precision values of the combinations of the four objectives *Maximize Requirements Coverage* (RC), *Maximize Failure History* (FH), *Maximize Business Relevance* (BR) and *Maximize Failure Probability* (FP) in addition to the mandatory *Minimize Test Set Size* (TSS) objective. The bar chart shows the resulting score on the Y-axis and the combinations leading to these scores on the X-axis. The presented results are the averages of ten repetitions of the GA, aggregating the precision of the computed Pareto-optimal solutions for each objective combination.

The bar chart shows that the overall obtained precision is low. This is due the fact, that only a small subset of failures can be found in the system. In average, test sets contain about 50% of all test cases, i.e., there is still a considerable number

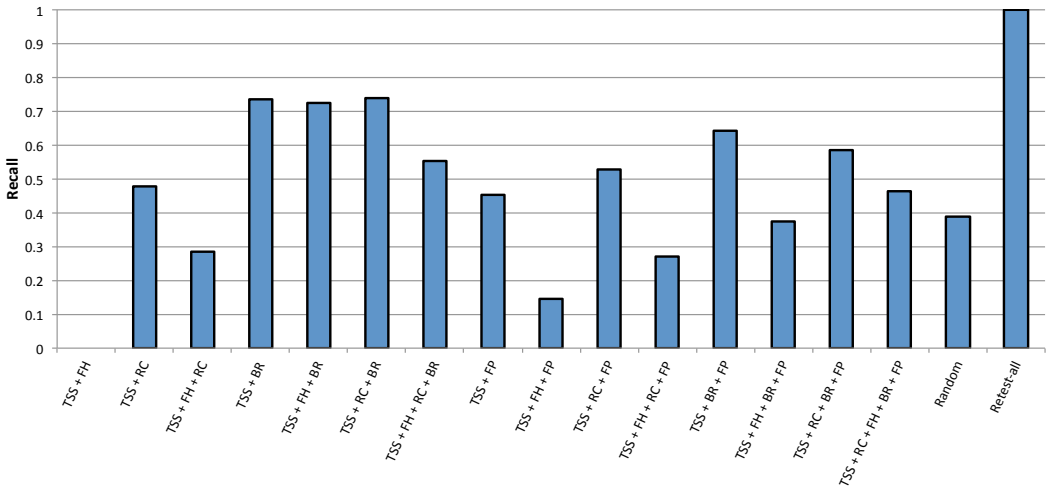

Figure 6.8: Aggregated Recall Values for BCS

of test cases compared to all failures in the system (cf. Appendix B for precise test set sizes). The highest average precision of 0.12 has been achieved using the BR objective for BCS in combination with mandatory TSS. In contrast, the FH and TSS combination did not reveal any failures as the average test set size was only two test cases, which probably contained test cases with a failure history. For BCS, each test case only reveals one failure, i.e., the history of a test case was not useful to find new failures.

Retest-all did not achieve a high precision, as it selected all 127 test cases, only revealing 7 failures. Thus, the precision value is 0.055 and less than half of the best achieved precision by our test case selection technique. The random test case selection performs even worse with an average precision of 0.042. This shows that our test case selection technique is superior to retest-all and random selection for BCS in terms of precision as it finds a high amount of failures compared to the selected test cases.

Recall Values for BCS. We measure the recall of our test case selection compared to the random selection and retest-all. The recall values obtained for the 15 different objective combinations as well as random and retest-all selection strategies are shown in Figure 6.8.

By definition, the retest-all approach achieves the best recall value of exactly 1 as it selects all test cases, i.e., it inevitably selects all test cases which will find a failure. Our multi-objective selection is able to achieve an recall value greater than 0.7 for three objective combinations: $\{TSS, BR, TSS\}$, $\{FH, BR\}$ and $\{TSS, RC, BR\}$. It is evident that the business relevance objective has a positive impact and is present in all three combinations. This shows the relevance of risk-related objectives in re-

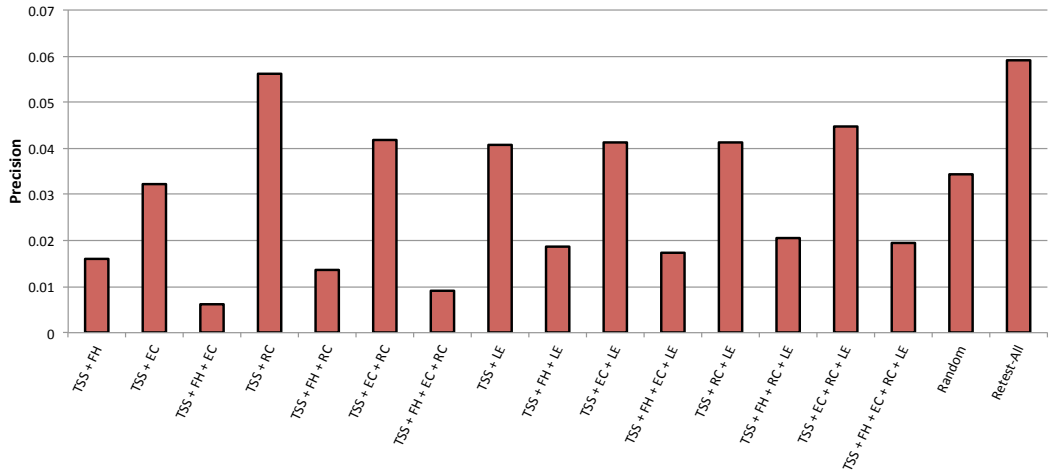

Figure 6.9: Aggregated Precision Values for Industry Data

gression testing. Again, the worst result is achieved by the $\{TSS, FH\}$ combination as it does not reveal any failures in the particular scenario and, thus, leads to a recall value of 0. The random test case selection achieves an insufficient average recall value of 0.39.

The results show that our selection technique was able to improve precision and recall values compared to a random approach. Precision has also been improved compared to retest-all, while the optimal recall value is not achieved.

Precision Values for Industry Data. Analogue to BCS, we compare the precision of the different objective combinations executed on industry data. Results are shown in Figure 6.9 as bar chart. The highest precision value of 0.056 has been achieved using the $\{TSS, RC\}$ objective combination. This precision is lower than the precision of the retest-all approach, which achieves a precision value of 0.059. In contrast, the random selection only leads to a precision of 0.034. Looking at the bar chart in Figure 6.9, it is noticeable that combinations using the *Maximize Failure History* (FH) objective have inferior precision values. Again, this is due the fact that each test case was linked only to one failure and, thus, did not reveal any new failures when selected. However, selecting and executing test cases which revealed failures in the past is still important for regression testing to ensure bug fixes.

Recall Values for Industry Data. We evaluated the recall values for the different objective combinations for industry data. The resulting aggregated values of all Pareto solutions for each objective combination are shown in Figure 6.10. There are many close contenders for the best recall values for industry data. The highest value is achieved using the four objectives $O_{selected} = \{TSS, FH, RC, LE\}$. This combination leads to a recall value of 0.472. Random selection is able to achieve

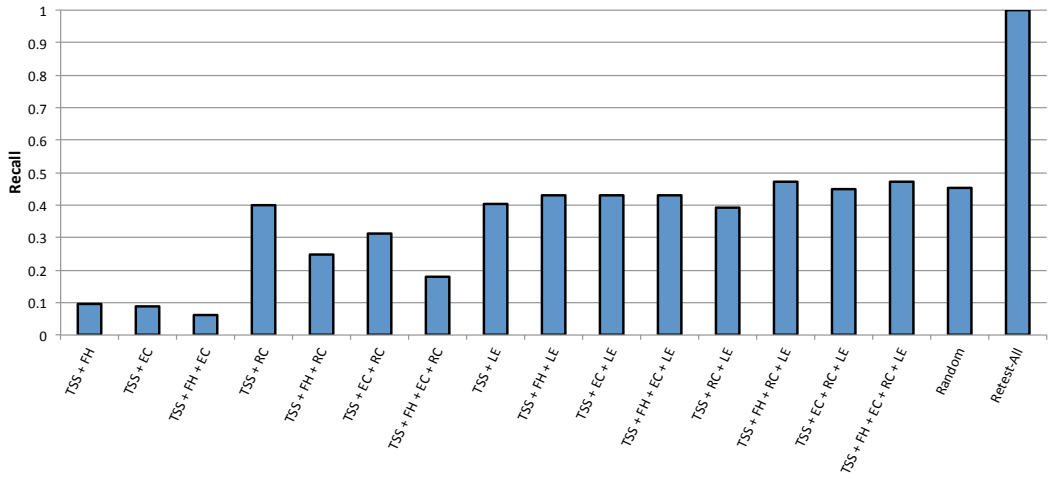

Figure 6.10: Aggregated Recall Values for Industry Data

a value of 0.453. Analogue to BCS, retest-all achieves the best possible value of 1, as it selects all test cases and finds all failures eventually. This shows that the selection of more than two objectives can improve the test case selection quality, i.e., a many-objective approach supports black-box test case selection.

Discussion of Applicability. Our test case selection technique was able to improve precision and recall compared to a random approach for BCS and industry data. While the retest-all approach obtains the optimal recall value, it lacks in terms of precision, which becomes especially evident for BCS, where our test case selection technique outperformed retest-all in this regard. For real-world data, retest-all achieves a slightly better precision. However, retest-all is flawed as it requires the execution of all test cases to achieve these values. In contrast, our test case selection is able to reduce the number of test cases to be executed by more than 50%. A detailed overview of the test set sizes for all objective combinations for BCS and industry data, is shown in Figures B.1 and B.2 in Appendix B.

Consequently, our test case selection technique is able to either outperform or achieve a similar selection quality as both, random selection and retest-all for both subject systems. In addition, our regression test case selection approach reduces the test set size significantly compared to all test cases, which makes it superior compared to a random or retest-all testing approach.

Pareto Front Distribution. To gain more insight in the actual results of our regression test case selection technique, we analyze the best Pareto solutions for both, BCS and industry data. For BCS, the best objective combination $\{TSS, BR\}$ is used. To assess the quality of Pareto optimal solutions, we compute the F-scores for each Pareto-optimal test set generated by the best objective combination. For

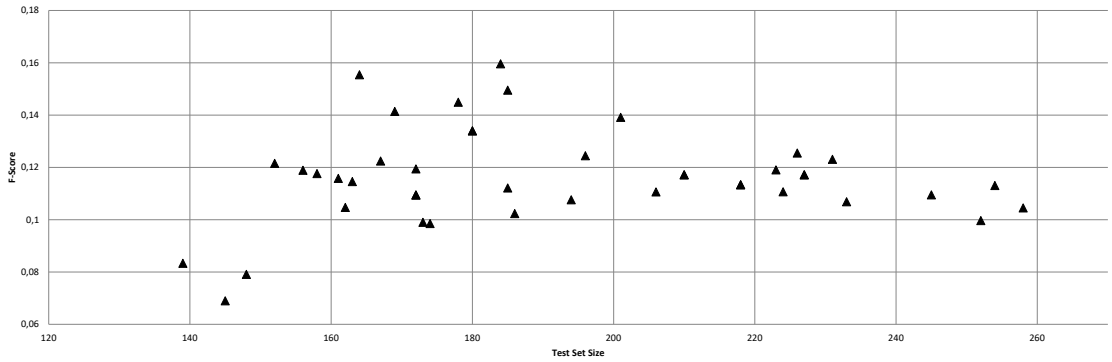

Figure 6.11: Distribution of Test Set Size and F-Score for Best Pareto Front for Industry Data ($O_{selected} = \{TSS, RC\}$)

two or more objectives to be optimized, the Pareto Front usually consisted of 100 individuals. We are interested in how the test set size of Pareto solutions correlates with the test sets quality in terms of F-score and if there is a sweet spot achieving the best results. We show the Pareto front of the $\{TSS, RC\}$ objective combination for industry data in Figure 6.11, which constitutes the best overall result. The x-axis constitutes the test set size and the ordinate represents the achieved F-score.

As shown, the smallest test set size is about 140 of the 577 test cases and the largest Pareto-optimal test set comprises nearly 260 test cases. The differences in F-score are large between different Pareto solutions. The best solution achieves an F-score of 0.16, whereas the worst only achieves an F-score value of 0.07. We are able to observe that the best solutions are grouped around $25-35\%$ of the total test set size. This concludes that our regression test case selection approach performs best compared to retest-all as it reduces the number of test cases but keeps the selection quality similarly high.

For BCS, an overview of the solutions is given in Figure B.3 in Appendix B. We observe a similar distribution of Pareto-optimal test sets, with an optimum around 30% of selected test cases.

RQ2: *How* efficient *and* adaptable *is our regression test case selection technique, i.e., is it is applicable to incomplete artifacts?*

We answer the second research question regarding the *efficiency* and *applicability* of our test case selection technique in the following, using both subject systems. First, we examine if our test case selection technique is efficient enough to be applied to different types of systems in recurring regression testing scenarios. Second,

Table 6.3: Average Computation Times of Test Case Selection

	BCS	**Industry**
Test Cases ($\|\mathcal{TC}_{sys}\|$)	127	577
Random	0.25 ms	3.8 ms
Fastest Comp. Time	150 ms ({FH})	300 ms ({FH})
Average Comp.Time	195 ms	9,200 ms
Longest Comp. Time	255 ms ({FH,RC,BR,FP})	18,500 ms ({FH,RC,LE,EC})

we apply the technique using the incomplete data, i.e., we perform all objective combinations and assess, if they produce statistically significant output.

Assessing Test Case Selection Efficiency. We investigate the efficiency of our test case selection approach. If our test case selection is too computationally expensive, it is not suited to be applied to real-world software testing projects. Thus, we measure the time it takes to compute random orders and the time it takes to perform our test case selection technique and compare the results. We do not measure the manual test case selection duration as it is very subjective and time-consuming. Table 6.3 shows the measured computation times of the different test case selection approaches for both subject systems.

The results indicate that both, the random and our test case selection technique are very efficient. Due to its simplistic nature, the random approach is the fastest in computing test sets with an average computation time of 0.25 ms for BCS and 3.8 ms for industrial data. Still, our test case selection technique is able to achieve a high efficiency, requiring 195 ms on average to find the Pareto optimal solutions for BCS and 9,200 ms for real-world data. The fastest computation is achieved using single objectives, in particular the *Failure History* (FH) with 150 ms for BCS and only 300 ms for industry data. Analogous, the more objective the optimization comprises, the longer the GA takes to find a solution. While the differences in computation time for BCS are small (max = 255 ms), the longest computation for industry data takes 18,500 ms to finish. While the test set optimization for industrial data takes about 50 times longer than for BCS, the input size is only about three times as large. We cannot fully explain the large differences, but assume that the optimization of larger individual sizes increase the complexity of the GA optimization. The basic runtime of a GA is $O(n \cdot pop)$, where n is the number of iterations chosen and pop the size of the population as each individual is manipulated (several times) in each iteration. This is the same for both case studies as we applied the same GA with the same number of iterations (5,000) and the same population size (100). One clue

for the differences in efficiency are the applied objectives. For industry data, we applied the *Maximize Last Execution* and *Minimize Test Case Costs* objectives, the latter being a more complex objective than others as it uses no fixed scale as input, but the actual number of seconds it took a test case to be executed.

Despite the differences in computation times, we argue that our test case selection technique is still very efficient as the overall computation for real-world data only takes mere seconds using up to five different objectives.

Statistical Significance Test. To ensure our test case selection techniques applicability to incomplete data, we execute the GA using all combinations of available objectives for both subject systems. To assess the flexibility of our test case selection technique, we analyze if the different results obtained by the objective combinations are statistically significant. Therefore, we perform the *Mann-Whitney-U* test on the result distributions of the objective combinations [MW47]. We apply this particular test as the distribution of all possible data samples is unknown [RG99]. We analyze the different result distributions of each objective combination with each other combination to compute the probability that two different objective combinations lead to the same underlying distribution. To compute the resulting *p-values*, we perform a total of 105 tests for each of the two subject systems. We aim to identify if combinations exist, which are likely to generate the same result distribution, i.e., there is no significant difference between the underlying distributions of the results. As significance threshold we use the alpha value of 0.05, i.e., distributions, which are not statistically significant yield a p-value greater than this threshold.

The results of our significance test reveal, that for BCS 98% and for industry data 93% of the combination comparisons are statistical significant, i.e., they result in a p-value ≤ 0.05. However, there were certain combinations which failed the significance test. These combinations are listed in Table 6.4 together with the computed p-value.

As the table indicates, only a few comparisons are statistically insignificant. For BCS, only *two* comparisons failed the threshold of $p \leq 0.05$. For industry data, a total of *seven* comparisons led to non-significant distributions between them. Most of these comparisons contain either the execution cost or last execution time, which seem to correlate in their result distributions. However, due to the high number of significantly different distributions in the evaluation (98% for BCS and 93% for industry data), we argue that our test case selection technique results in different result distributions depending on the selected objectives. Thus, our search-based regression test case selection technique is suited to be applied to different software domains, being able to handle different data input. This makes it more flexible compared to existing greedy techniques, which are restricted to a fixed set of data.

Summarizing, we argue that our test case selection approach is effective, efficient and flexible for incomplete data and different software systems.

Table 6.4: Non-Significant Comparisons of F-Score Distributions

Subject System	First Combination	Second Combination	P-Value
BCS	{TSS, EC}	{TSS, RC, BR, FP}	0.058
	{TSS, FH, BR}	{TSS, RC, FP}	0.511
Industry Data	{TSS, EC}	{TSS, FH, RC, LE}	0.083
	{TSS, EC, RC}	{TSS, EC, LE}	0.191
	{TSS, LE}	{TSS, RC, LE}	0.348
	{TSS, EC, LE}	{TSS, RC, LE}	0.418
	{TSS, EC, RC}	{TSS, RC, LE}	0.578
	{TSS, EC, RC}	{TSS, LE}	0.832
	{TSS, FH}	{TSS, FH, RC}	0.856

TSS = Minimize Test Set Size, **EC** = Minimize Execution Costs, **RC** = Maximize Requirements Coverage, **BR** = Maximize Business Relevance, **FP** = Maximize Failure Probability, **LE** = Maximize Last Execution, **FH** = Maximize Failure History

6.3.6 Threats to Validity

Referring to Runeson and Höst [RH09], we identify the following internal and external threats to validity for our evaluation of the proposed multi-objective test case selection for black-box software versions.

Internal Threats to Validity. The random nature of GAs poses an internal threat to validity as results are non-deterministically computed and outliers can reduce the confidence in results [Dav87, KCS06]. To tackle this issue, we repeated our analysis of the multi-objective regression test case selection ten times for two different subject systems, mitigating the effects of the search-based nature of the technique. To further increase the confidence in our results, we computed the mean, standard deviation (SD) and variance between the F-score results of the ten different repetitions of our test case selection technique. The SD is between 0.011 and 0.021 for industrial data and $0.02 - 0.05$ for BCS. All measured standard deviation values are rather low, i.e., our test case selection technique is stable and presenting average values should mitigate any outliers. The detailed statistical results for both subject systems are shown in Table B.1 in Appendix B.

Another internal threat to validity is the algorithm we selected for evaluation, namely NSGA-II. Other GA techniques might scale better using more than two objectives, i.e., many-objective algorithms could suit the problem better and potentially result in better results [LLTY15]. However, NSGA-II has proven to be a good solution and is a standard algorithm used for different types of multi-objective optimization problems [SA13]. It has shown promising results for our problem statement of black-box test case selection.

External Threats to Validity. One industrial case study is not enough to show the general applicability of our test case selection technique on real-world systems, even though our test case selection approach performed well for our industrial subject system. However, using two different case studies and applying different objectives and objective combinations shows the flexibility of our test case selection technique. In addition, the assessment indicates that our test case selection approach is able to achieve good results for different types of systems. While the risk-based information was not available in our industrial data, the corresponding objectives have shown great results for BCS when selected. However, further case studies have to be performed to assess the potential of risk-based objectives using real-world data. We did not compare our test case selection technique to existing test case selection techniques besides random and retest-all as other technique requires either different data or tools were not available. Future case studies should compare our test case selection to existing techniques to further assess the quality and applicability of our black-box test case selection technique.

6.4 Related Work

Search-based software testing (SBST) is a popular research direction, which is concerned with the optimization of testing based on meta-heuristic search techniques such as genetic algorithms [XES+92, HM10, McM11]. SBST has been adapted into different aspects of testing due its generic nature, such as structural testing [BSS02], testing of non-functional system properties [ATF09] or mutation testing [JH08]. In fact, Harman et al. [HJZ15] notice a polynomial yearly rise in publications in the field of SBST. As this thesis focuses on regression testing, we examine and present related work, which uses SBST for regression testing, i.e., selection or prioritization of test cases. In addition, we inspect techniques that are related in terms of available data and system-level regression testing. Table 6.5 shows an overview of related techniques and classifies them according to the type of technique and required input data. Similar to previous chapters, relevant criteria are marked with "X", however, if the paper does only partially support a data artifact (e.g., only business relevance but not failure probability for risk) we indicate this using the notation "(X)".

Single-Objective Black-Box Test Case Selection. Some of the first and most prominent black-box test case selection techniques are found in the domain of model-based testing (MBT) [FvBK+91, FIMN07, NZR09, HAB11]. However, MBT requires test models (e.g., state machines) which represent the specification of the system. While these allow for an elegant solution in black-box testing, test models often do not exist in practice. Hence, we focus on related work which can cope with black-box data similar to our test case selection approach or use similar approaches, even though some partially use MBT-related artifacts.

Table 6.5: Categorization of Related Work for Black-Box Test Case Selection

Related Work	Technique		Used Artifacts						
	Selection	**SBST**	**RC**	**FH**	**Risk**	**Cost**	**Age**	**Code**	**MBT**	**Other**
Single-Objective Black-Box Test Case Selection										
Chen et al. [CPS02]	X								X	
Chittimalli and Harrold [CH08]	X		X		(X)					
Ekelund and Engström [EE15]	X			X				X		
Hemmati et al. [HAB13]	X	X							X	
Herzig et al. [HGCM15]	X			X		X				
Rogstad et al. [RBT13]	X	X							X	
Zheng et al. [ZWRS07]	X							X		X
Multi-Objective Regression Testing										
Anwar and Ahsan [AA13]	X		X	X	(X)	X				
Briand et al. [BLC13]		X				X			X	
De Souza et al. [dSdMPdB11, dSPdAB14]	X	X	X			X				
Epitropakis et al. [EYHB15]	X	X		(X)				X		
Huang et al. [HGLJ13]	X	X				X		X		
Mondal et al. [MHD15]	X	X						X	X	
Yoo and Harman [YH07a]	X	X		X		X		X		
Our Test Case Selection Technique	X	X	X	X	X	X	X			

Selection = Test case selection approach, **SBST** = Search-based testing technique, **RC** = Requirements coverage, **FH** = Failure History, **Risk** = Risk information, **Age** = Test case age, **MBT** = Model-based testing data

Chen et al. [CPS02] describe a specification-based test case selection. They assume that a specification is given as UML activity diagram [OMG15]. To select test cases, they use the risk analysis model presented by Amland [Aml00]. They separate test cases into two categories, *targeted test cases* (i.e., execute important requirements) and *safety test cases* (i.e., selected to ensure defined coverage). Compared to this work, the approach is model-based and does not apply search-based testing and other meta-data. **Chittimalli and Harrold** [CH08] introduce a requirements-based test case selection technique. It takes the criticality of requirements into account, which is derived by their covered functionality and their complexity. In contrast to our work, the authors assume to know about changes between programs p_{orig} and the successor p_{mod}. They compute a requirements traceability matrix to find test cases which are required to cover modified requirements between programs. The authors evaluated their technique, showing promising results in terms of test effort reduction. **Ekelund and Engström** [EE15] present a history-based test case selection. They extract differences from different test runs and compute regression testing recommendations. Their Difference Engine tool correlates code with test cases at package level and, thus, is not solely black-box. Similar to our work, the authors compute precision and recall and are able to improve testing efficiency compared to retest-all. **Hemmati et al.** [HAB13] perform similarity-based test case selection for model-based testing. The authors found the *(1+1) evolutionary algorithm* [DJW02] to be the best algorithm for the similarity-based test case selection. This type of GA restricts the population size to contain only one individual at the time. The authors also compared different types of selections. The results of their selection imply that the similarity partition-based technique performs best in terms of fault detection rate, and that the evolutionary algorithm outperforms a greedy selection. Compared to our test case selection approach, they focus on similarity-based testing and use test models as foundation. **Herzig et al.** [HGCM15] present THEO, which is a generic test selection technique based on cost models. Their approach evaluates if the expected costs of test cases exceed a certain threshold according to available resources. Test cases are skipped if the costs of running a test case exceed the costs of not executing it. Costs are based on the historical records of a test case's failure finding capabilities and the cost to actual run the test case based on the machines and engineers involved. In contrast to our work, THEO is solely based on cost models and historical test data. **Rogstad et al.** [RBT13] present a selection technique for database applications. They assume black-box knowledge and apply a similarity-based approach for test case selection. The authors combine two techniques, classification tree models and similarity-based test case selection, and evaluate how the combinations perform compared to a greedy technique. Classification tree models partition the input domain of the SUT, which allows the automatic generation of test cases in black-box domains [LW00]. Similar to Hem-

mati et al. [HAB13], the authors applied the *(1+1) evolutionary algorithm* [DJW02] to achieve the selection. In contrast to our test case selection approach, they use classification trees and do not assume other black-box criteria, such as risk-based testing. **Zheng et al.** [ZWRS07] present the I-BACCI black-box test case selection approach. They analyze the binary code of the program, finding changes between two program versions. Based on these changes, the authors identify test cases which cover affected functions. Even though the approach is of black-box nature, it is based on code analysis and does not incorporate any of the meta-data used in this thesis.

Multi-Objective Regression Testing. Only few authors have investigated possibilities to use multi-objective algorithms to improve regression testing. **Harman** [Har11] states that multi-objective regression test optimization is an important area to investigate. He states that objectives are either *values* to be maximized or *costs* to be minimized. While he does not describe a specific approach, he gives examples for potential optimization objectives, which contain some of the objectives used in our test case selection, i.e., test execution costs, requirements coverage fault history or business sensitive objectives.

Anwar and Ahsan [AA13] propose an approach to optimize test suits, which is similar to our test case selection technique. They define four objectives to be optimized based on fuzzy logic. While their approach is not search-based, they also consider requirements coverage, execution costs and failure history. In addition, they consider the *requirements failure impact*, which is similar to our business relevance. However, they do not consider a minimize test suite, maximize failure probability or last execution date of test cases. Furthermore, they failure severity is not considered for their approach. **Briand et al.** [BLC13] present a multi-objective test case prioritization technique for state-based test cases. In their work, test cases are derived using the transition tree method [Bin99]. The prioritization is based on the data-flow of test cases, i.e., they present a model-based approach. To compute the priority of test cases, they define four objectives to be optimized by an GA, namely SPEA2 [ZLT01]. The objectives are based on cost (number of transitions triggered) and different data-flow goals, such as the number of definitions covered by a path normalized over the number of all definitions. The result is an ordering of transition tree paths with the goal to intend defects as early as possible. The evaluation shows that the resulting test case orders increase the data-flow coverage and, thus, the fault detection rate compared to random orderings. In contrast to our test case selection approach, they use GA to prioritize test cases based on data-flow analysis of transition trees. **De Souza et al.** [dSdMPdB11, dSPdAB14] present a test case selection technique based on particle swarm optimization [KE95]. They consider two objectives at once: minimizing execution costs (time) and maximizing requirements coverage of test cases. Results indicate that their technique outperforms random

selection. We consider more objectives to be optimized at once for our test case selection approach. **Epitropakis et al.** [EYHB15] present a multi-objective test case prioritization technique using evolutionary algorithms. They consider three objectives: *average percentage of code coverage*, *average percentage of changed code covered* and *average percentage of past fault coverage*. Trivially, these objectives are code-dependent, which makes them different from our test case selection approach. They evaluate their prioritization based on six different systems. Their approach shows similar or better results in terms of the cost-aware $APFD_c$ metric [EMR01] compared to greedy techniques. **Huang et al.** [HGLJ13] present a test case selection using an "improved genetic algorithm". They adapt the GA in a way, that the inbreeding of individuals is reduced and the problem of sticking to local optima is avoided., e.g., by adding an excellent gene pool used in the mutation step to replace bad genes in current individuals. They also add "different" individuals using a greedy algorithm, when the dissimilarity of existing individuals is below a certain threshold. For test case selection, they optimize two objectives: *Sensitivity Function Test Coverage* and *Test Cost*. Due to the first objective, which requires information about the functions covered by test cases, the approach is of white-box nature. **Mondal et al.** [MHD15] introduce a multi-objective test case selection technique based on code-level information. In particular, they optimize both, code-coverage and test case diversity. Similar to this work, they apply the NSGA-II [DPAM02] algorithm. **Yoo and Harman** [YH07a] present a multi-objective test case selection approach. They also apply NSGA-II to realize their approach [DPAM02]. Moreover, they also describe a fault-history and execution cost objective to be maximized. However, their also assume to have knowledge about the code of the SUT, to maximize code coverage.

Summarizing, our analysis of related work (cf. Table 6.5) shows that our test case selection technique is indeed a novel approach for black-box regression testing as it uses a unique combination of black-box objectives as input for a multi-objective GA. However, only two techniques use risk-related information, even though not to the extend of our test case selection approach. Of the presented multi-objective techniques, five out of seven use test case costs as objective, which seems to be a popular choice.

6.5 Chapter Summary and Future Work

Summary. In this chapter, we introduced a novel multi-objective test case selection technique for regression testing. While multi-objective test case selection has been investigated in the past, it has never been analyzed for black-box systems in the complexity and flexibility as proposed in this thesis. We defined seven different black-box objectives to be optimized by a GA: *Minimize Test Set Size*, *Maximize*

Requirements Coverage, *Maximize Business Relevance*, *Maximize Failure-probability*, *Minimize Execution Costs*, *Maximize Failure History* and *Maximize Last Test Execution*. Because of the nature of test case selection, we decided to always select the *Minimize Test Set Size* objective in combination with all other objectives. The other objectives can be combined arbitrarily, which allows for a more flexible test case selection technique compared to single-objective approaches as we can adapt the optimization to the available data. In particular, we subsume different approaches pursued in the past as greedy or single-objective strategies like *history-based*, *cost-based* or *risk-based* test case selection.

We investigated the feasibility of our test case selection approach by analyzing the applicability of our test case selection technique to two case studies, BCS and real-world industry data. We investigated the *effectiveness* of our test case selection technique by measuring precision, recall and F-score for all objective combinations. To ensure that these results are reliable, we repeated our experiments ten times and performed the *Mann-Whitney-U* significance test [MW47], which shows that different objective combinations do indeed reveal different results in nearly all cases. We compared our multi-objective test case selection approach to a normalized random selection and the retest-all approach. Results indicate that our test case selection technique is able to outperform a random selection by far. While retest-all has a similar quality for industrial data, it requires all test cases to be executed. In contrast, our test case selection reduces the test set size by at least 50% in average. Moreover, we identified a correlation between the test size reduction and selection quality. Finally, the technique has been shown to be efficient.

Future Work. While our evaluation shows promising results for two subject systems, more case studies have to be performed to allow for a generalization of our findings. In particular, more industrial data has to be examined, especially when given risk-based information, as we lack this information for our real-world system. While Henard et al. [HPH+16] found in a study that white-box and black-box test case prioritizations are able to perform very similarly, such a comparison is currently missing for test case selection. Hence, it is of interest to analyze how a combination of white-box and black-box data influences the overall results. Buchgeher et al. [BERL13] ascertained that a test case selection based solely on code coverage information leads to a large set of potential test cases, i.e., further knowledge is required in addition to code-coverage. This shows the potential of the combination of our current objectives with white-box related objectives such as maximizing code coverage to improve test case selection quality. Hence, we assume that a more fine-granular test case selection might be able to improve the selection quality even further. In addition, our flexible approach allows to incorporate different code-level coverage criteria such as statement or branch coverage [LV04].

Applying other multi-objective algorithms besides NSGA-II is also an important

task for future work, to assess which GA are suited to solve multi-objective test case selection for system-level testing. Especially when handling more than four objectives at once, *many-objective algorithms* might solve the underlying optimization problems better [LLTY15]. In the same regard, other types of mutation or crossover operators can be used to find different optimization strategies for test case selection. We also see possibilities to adapt the selection technique into a prioritization approach. This might be more desirable if resources for regression testing are very limited or unknown a priori, as testing can stop at any time in prioritization while ensuring that the most important test cases have been executed. However, transforming the technique into a prioritization approach requires different mutation and crossover operators, as the order of genes has to be changed.

7 Machine Learning-based Test Case Prioritization for Software Versions

The content of this chapter is largely based on the work published in [LSN+16].

Contribution

We propose a black-box test case prioritization approach for software versions based on supervised machine learning. Our test case prioritization approach emulates decisions and experiences made by test experts. Test cases are described in natural language and a manually selected subset is provided as training data for a machine learning technique. The result are prioritized test cases, which are able to find system failures early.

As described in Chapter 6, regression testing is a major task in the software life cycle, as software development is not one singular process that is finished after a software system is released, but often continues after the initial release to comprise bug fixes, feature extensions or new versions [RH97]. For system-level testing, regression testing often is performed manually in black-box environments, as the system is tested as a whole on user level [Sne07]. Due to restricted resources, test experts have to make decisions on what to test in system-level regression testing on a regular basis. In case of black-box testing, intrinsic expert knowledge is applied to identify test cases, which have a high likelihood to detect failures or which contribute to specification coverage. Expert knowledge is derived from previous test runs, experiences from prior projects or information about changes in the software system, but is not documented or formalized. Thus, this knowledge differs from one test expert to the next and is of a very subjective nature.

Test case prioritization has been proposed as approach to improve regression testing [YH07b]. A prioritization of test cases allows to continue testing until resources are exhausted, while still ensuring that the most important test cases have been executed. However, current black-box regression test cases are selected and not prioritized as human testers are only able to make up to 100 pairwise comparisons of test cases in a consistent fashion [ABPS05], which makes a manual prioritization of large test sets infeasible. The subtle differences and instinctive decisions made by human experts make an automatic analysis of expert knowledge in black-box testing a difficult task, which has not been investigated enough in computer science yet.

This thesis contributes a novel technique to provide an automatic black-box test case prioritization technique for a set of system test cases $\mathcal{TC}_{sys} = \{tc_1, \ldots, tc_n\}$. It supports regression testing of black-box software versions. Engström and Petersen [EP15] report that testing research often does not find its way into practice. Hence, our test case prioritization technique formalizes decisions and experiences made by test experts. We use a supervised machine learning algorithm to learn a prioritization model based on existing knowledge. Test cases are defined in natural language, describing test steps to be manually executed, and additionally assigned system-testing meta-data, such as linked requirements or revealed failures. Our technique supports the identification of similarities in the description of test cases to identify their importance. The model allows for the prioritization of arbitrary system test cases, described in natural language. Consequently, we introduce test case prioritization for real-world scenarios where source code is not available. The priority of a test case resembles the computed likelihood that the test case will detect failures in future executions. Thus, we aim to improve the failure finding rate for a set of test cases. Our evaluation shows that we are indeed able to improve regression testing for two subject systems, including real-life automotive data.

The remainder of this chapter is structured as follows: We first give a short introduction in supervised machine learning and, based on this, provide the concept of our test case prioritization approach. Afterwards, we explain and discuss our evaluation and the gathered results. Related work is described to show the novelty of our test case prioritization technique and the necessity to investigate this type of regression testing scenario.

7.1 Supervised Machine Learning

Machine learning comprises techniques which attempt to learn patterns or rules based on existing data [Bis06, WFH11]. This data is provided by an expert and used for *training* of the machine. Selecting training data is an essential step in machine learning [Bis06].

Training Data Selection. Providing training data influences the patterns that the machine learning detects and, thus, heavily influences the quality of the results of the technique. Two major types of machine learning techniques are distinguished based on the provided training data: *Supervised* and *unsupervised* learning [WFH11]. Supervised learning receives a set of labeled training data, i.e., each training entity is associated with a class label to which a particular data instance belongs to [Kot07]. Other entities are then to be assigned to a class, according to the label distribution in the training set and the patterns recognized by the chosen algorithm. Several techniques exist to realize such a classification of data [Kot07, WFH11]. A commonality of supervised techniques is that an expert has to provide class labels for

the training data instances. In contrast, unsupervised learning has no information about class labels and, thus, has to identify these classes based on the characteristics of the data instances without additional knowledge. In this thesis, we solely use supervised learning as our goal is to adapt expert knowledge of testers and, thus, we aim to imitate their decisions which they provide in form of labeled training data.

To achieve the best results of ML classification, certain quality aspects have been derived for a sufficient selection of training data [GKN12], which we assume to be satisfied when applying our test case prioritization technique:

- **Class Balance:** Supervised learning requires labeled training data. For training data selection, it is important to keep a certain level of balance between the different classes provided in the training set. For example, in a binary classification, two labels *+1* and *-1* can be used. The training data should provide sets for both classes of similar size. A *class imbalance* results in retrieved models which might be biased in terms of data distribution and reliability. For example, the model might be overfitted for one class-label and not detailed enough for another class.

  Weiss [Wei04] shows in a survey on class imbalance that two main types of techniques exist to cope with class imbalance, namely *data sampling* and *boosting*. Data sampling either increases or reduces the data instances for certain classes, such that the final data set is balanced. One simple form would be random sampling or more sophisticated algorithms, which have been described in the literature [BVSF04]. Boosting, on the other hand, has not specifically been designed to solve class imbalance, but shows good results nonetheless, as the performance of weak classifiers is increased, which has a positive impact on the classification quality [GKN12].

- **Number of Entities:** The size of the training data set is of importance as well and correlated with the class imbalance problem. If the number of provided data instances is too large, the classification model might be *overfitted* or the computation of a solution becomes very complex [Die95]. On the other hand, too few data instances reduce the robustness of the model and the algorithm might not be able to deduce certain patterns and rules for the given classes [BPM04]. Thus, a right amount of data is too be chosen, which is non-trivial and requires domain knowledge and experience.

- **Feature Dimensionality:** The classification is based on different features which represent the entities to be classified. Each data instance is a feature set, representing a point in an n-dimensional vector-room for n features. Thus, the computation heavily depends on the feature-complexity. The more features are used for data classification, the more complex the computation becomes.

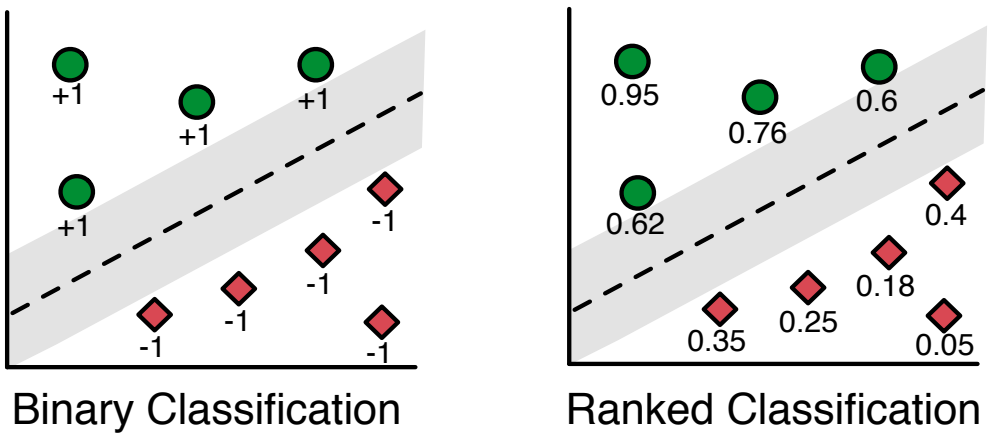

Figure 7.1: Binary vs. Ranked Classification Models

As a result, too complex or too many features should be avoided [GKN12]. Several techniques have been introduced to cope with this problem, such as filter-based feature selection [KGN12]. In case of this work, the set of features becomes very large as we analyze the occurrences of words in a test cases. However, most of the values are 0, as only a small subset of all words will occur in each test case. Thus, we prefer machine learning algorithms that can cope with sparse data.

Consequently, certain preparations might be necessary to reduce the risk of class imbalance and improve the test case prioritization. To avoid the described issues, training data should always be selected by an expert.

Supervised Machine Learning Techniques. There exist plenty techniques in supervised learning which are applicable for classification [Kot07, WFH11]. However, some restrictions have to be made when selecting an algorithm for test case prioritization. First and foremost, the classification technique shall produce a *ranked classification model* (also described as regression model), which differs from typical discrete classification models in some regards. Figure 7.1 shows an example to illustrate the differences between these two types of classifications using a *Support Vector Machine* (SVM), which is a popular type of machine learning algorithm used for classification and regression tasks [CST00, WFH11].

On the left hand side of Figure 7.1, a discrete binary classification is shown, which is a standard machine learning task [WFH11]. It assigns a fixed class to each data instance, according to a learned model. Green circles represent data instances of the *+1* class, while red squares are *-1* labeled data points. These data instances have been labeled by an expert and selected as training data set. The axes represent different features. In case of the example only two features are available to classify

data points. A learned classification model is represented by the dotted line, also called *hyperplane* for higher feature spaces. The grey area around the hyperplane is the *margin* of this model and describes the distance to the nearest data points, referred to as *support vectors* in SVMs [CST00]. The goal of SVMs is to maximize the margin between the nearest support vectors to improve the reliability of the classification task. This represents a straight forward model representation, as the data points, which belong to different classes are clearly separated from each other. In context of this thesis, the classes resemble *to test* and *not to test*, and the data points are test cases.

A binary classification as shown on the left side of Figure 7.1 would lead to a test case selection, as test cases are assigned to one of the two classes. As we are interested in test case prioritization, a binary classification is not sufficient. Instead, we implement a ranked classification technique which computes a ranking function to assign weight values to data instances, as shown on the right-hand side in Figure 7.1. The prioritization approach is not restricted to one particular ML technique, examples for applicable techniques are *relevance vector machines* [Tip01] or *ranked support vector machines* [Joa02], but potentially also probabilistic techniques, such as Bayesian networks [FGG97]. The learned classifiers are used to rank new data instances according to their computed probability to be in one class or the other.

7.2 System-Level Prioritization Approach

The concept of the machine learning based test case prioritization for black-box versions is shown in Figure 7.2. The goal is to compute a prioritization function to prioritize system test cases $tc \in \mathcal{TC}_{sys}$. It consists of six different steps, which can be grouped into three major phases: *Data Collection and Preparation*, *Learning and Classification* and *Execution and Reporting*. We explain these phases in detail in the following.

7.2.1 Data Collection and Preparation

Input Artifacts. To apply our test case prioritization approach, available data has to be prepared for analysis. Hence, certain testing artifacts and properties are necessary to fulfill the prerequisites for our test case prioritization technique. An overview of the required information has been described in Chapter 3.1.2. Basically, we assume to have knowledge about *requirements*, *system test cases* and *revealed failures* in natural language. In addition, we assume that traceability between these artifacts is given and that black-box meta-data is assigned, e.g., test execution time or failure severity.

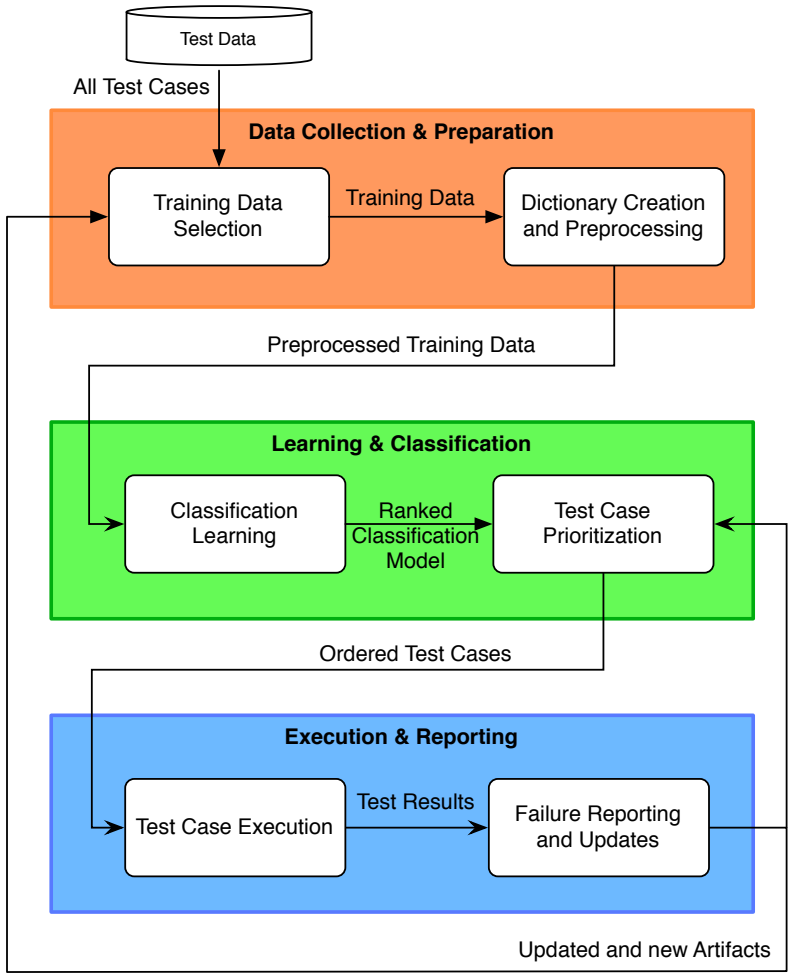

Figure 7.2: The Machine Learning-based Prioritization Technique (cf. [LSN+16])

The described data artifacts only resembles a baseline on which the test case prioritization technique is able to perform. The data can be extended by several means, which might positively influence the test case prioritization quality. For example, source code information or test models could be available. In the following, we assume that the data described in Chapter 3.1.2 is available and sound, even though reality shows that keeping a clean, up-to-date and redundancy-free database is a common problem in large software projects.

Select Training Data. As we apply supervised learning, the first step is to select a suitable set of training data. In case of this thesis, we assume the classes to be *to test* and *not to test* for test cases, i.e., the training data consists of labeled test cases $TC_{label} \subseteq \mathcal{TC}$, where each test case $tc \in TC_{label}$ is labeled according to its

class. To achieve suitable results, the guidelines for training data selection presented in Chapter 7.1 are to be applied by test expert. Selecting training data is a crucial step for our test case prioritization technique, which we assume to improve over time with increasing experience. However, manual test case selection is easier than prioritization, which is why we argue that training data selection can be performed by test experts in a reasonable amount of time.

Example 7.1: Training Data

Consider that $|\mathcal{TC}_{sys}| = 1{,}000$. Each $tc \in \mathcal{TC}_{sys}$ is linked to at least one requirement $req \in \mathcal{REQ}$. When selecting training data, the test expert should avoid to use all test cases, which is exhaustive and time-consuming. Instead, 100 to 200 test cases should be labeled as *to test* or *not to test*. Positive test cases are those, which are often applied as regression test cases or which test the newest functionality. Negative test cases might have not revealed a fault in several past executions or cover functionality which has not been changed for several versions and runs very stable.

Preparing the Dictionary. After training data has been selected, the test cases are transformed into a feature vector representation, which is required by the ML algorithm. While the meta-data is directly accessible as discrete values (e.g., seconds for cost, integers for failure priority etc.), the natural language description has to be prepared in advance to be suitable as input for test case prioritization. Therefore, we apply *natural language processing* (NLP) techniques to reduce the amount of redundancy and ambiguity within the data [JM00]. NLP is a very complex field of study, containing lots of different algorithms and concepts. In this work, we perform three basic techniques to prepare the *test case description* (TCD) for dictionary computation, namely *tokenization*, *stemming* and *stop word removal*. Tokenization detects words within texts and removes symbols as we do not consider them to be relevant. Stemming reduces certain words to a common word stem, avoiding redundancy and ambiguity between words. This step is important for verbs, e.g., reducing the words "moving" and "moves" to "mov", giving both the same meaning. The last step is the removal of stop words, which requires a predefined stop word list. Stop words are often occurring words that do not add any additional meaning to the content of the sentence, but are required by the language's grammar. Prominent examples for stop words are linking words or articles. We only generate a dictionary based on test cases in the training data TC_{label}, as these describe features to learn the prioritization model and reduce the dictionary size compared to analyzing all test cases. In future additions of this work, dictionaries might be extended to requirements covered by the test cases.

Example 7.2: Test Case Processing via NLP

Consider the following sentence to be part of a test case description in natural language:
Pressing the power window button moves the window up.
Tokenization detects the different words in the sentence and extracts them, removing punctuation marks as well. This leads to:
Pressing | the | power | window | button | moves | the | window | up
Next, we are able to remove stop words, which have no further meaning, resulting in the following words:
Pressing | power | window | button | moves | window | up
Using stemming, these words are reduced to the following form:
press | power | window | button | mov | window | up
One representative of these words will now be added to the dictionary, if they are not already present. In the example, the word *window* will only be added once.

Computing Feature Vectors. Once the original data has been cleansed from unnecessary fragments and unique representatives for each word have been added to the dictionary for all test cases, we use this dictionary as a baseline for the TCD feature of the test case prioritization. In particular, we are now able to create a vector representation of each test case including the contained words and other meta-data. If a word does not occur for a test case, the entry is simply 0, otherwise it contains the number of occurrences. As the dictionary size can become rather large in real life data sets, the feature vectors will also become very large, as each word adds another feature dimension for the machine learning. However, many entries will be 0 for the test cases, as their descriptions only contain a small subset of words occurring in the complete dictionary. These vector representations will be later used to compute the ranked classification model.

Meta-data values are encoded in these feature vectors as well. For each requirement $req \in \mathcal{REQ}$, we create a feature which is either 0 if the test case is not linked to the particular requirement or 1 otherwise. For failures, we introduce three different features. One summarizes the number of failures a test case is linked to. The other represents the severity of all linked failures as sum. In the third, the age of failures is represented. In this context, we use a predefined scale for failure age as described in Chapter 3.1.2. This reduces the complexity of the feature, which reduces the computational effort required to learn a model.

Example 7.3: Representing Test Cases as Vectors

Assume that we want to represent the system test case presented in Example 7.2 as vector containing the features regarding the TCD. The dictionary for the system shall consist of the following words:
power | window | mov | up | down | stop | button | finger | protection

Now, the particular system test case described is represented as the vector $(1, 2, 1, 1, 0, 0, 1, 0, 0)$. The number of occurrences of a word in a test case represents the value for the particular word scalar in the vector.

7.2.2 Learning and Classification

Learning. After labeled training data has been selected, our test case prioritization technique is able to learn a *ranked classification model* based on the data and feature distribution. This phase correlates strongly with the chosen learning algorithm, i.e., what type of algorithm performs the learning. In any case, the output of the learning process should result in a model which can be used to prioritize system test cases. Some types of output models are easier to understand than others, which might be an influencing factor for the choice of a sufficient ML algorithm.

Classification. Once a ranked classification model has been successfully learned from the labeled training data, it can be used to classify a set of test cases. Of course, these test cases do not have to be part of the original training data, i.e., unlabeled data can be prioritized according to their feature representation. For instance, all available test cases can be prioritized by the technique. Each test case is assigned a value, which will be used for the prioritization. The higher the value, the higher is the priority of the associated test case. We suggest to select a subset of test cases to prioritize instead of choosing all test cases as input, as the test expert has some knowledge about test cases and the current status of the project. Hence, he might identify test cases which are of no interest at all and, thus, need no priority value assigned. This reduces the computational effort and might lead to faster results.

Repetitive Learning. As software projects evolve over time, the process of learning a ranked classification model should be repeated iteratively to stay up-to-date and to ensure a certain prioritization relevance according to the current status of the project. We suggest to repeat the learning process for each major update, which introduces new features to the system, as this will require an adaption of regression testing. However, testers do not always have to select new training data from scratch. Rather, they only need to add new test cases which might become important after the changes of the software and move some test cases from the

positive class to the negative class or discard them, to reduce the testing overhead. Of course, completely new classifiers can be created as well by selecting new training data. This requires a higher manual effort compared than reusing training data.

7.2.3 Execution and Reporting

After a prioritized list of test cases has been computed, the test cases are executed by testers. As of now, our test case prioritization technique is designed for manual test cases. Hence, test experts execute the test cases in order of their rankings, starting with the most important one. Testing continues until a certain test end criterion is reached. In practice, available testing resources define how long testing continues. However, other criteria such as requirements coverage or a certain percentage of executed test cases are suitable as well using our test case prioritization technique.

Considering Preconditions. The presented test case prioritization technique takes a list of test cases and computes their priority according to their features. While this is the main idea of test case prioritization and matches the desired priority computation, it leads to another challenge regarding the applicability of our test case prioritization approach in complex testing environments. In particular, in system-testing, test cases are grouped to form coherent testing scenarios to improve test efficiency in manual testing [Sne07]. A set of preconditions $\mathcal{PC} = \{pre_1, \ldots, pre_m\}$ define the set of states, in which a system has to be in to commence a particular test case. In manual testing, a tester has to prepare a test case's precondition, then execute the test case and afterwards check if the observed result equals the expected result. However, test cases have different pre- and postconditions, e.g., different hardware or user rights are used for different functionalities. If only one prioritized test case list is generated, preconditions might be shuffled in arbitrary ways. Currently, preconditions are ignored for test case prioritization and, thus, even though we execute the most effective test cases at first, the execution might not be very efficient, as the setup of each test case has to be created from scratch. Hence, the ordering is suboptimal in terms of manual test case execution.

Example 7.4: Prioritization without Preconditions

In Figure 7.3, the left hand side shows an unordered list of test cases. The notation is *pre ID post*, where *pre*, *post* $\in \mathbb{N}$ are precondition references and ID is the name of a test case. Thus, test case *tc5* requires the first precondition and stays in the same state after its execution. An example setting is a profile settings site in a web-client, which allows to change the user's address information. After the information has been updated, the user stays on the same page. Other test cases, e.g. *tc4*, might change the state of the program.

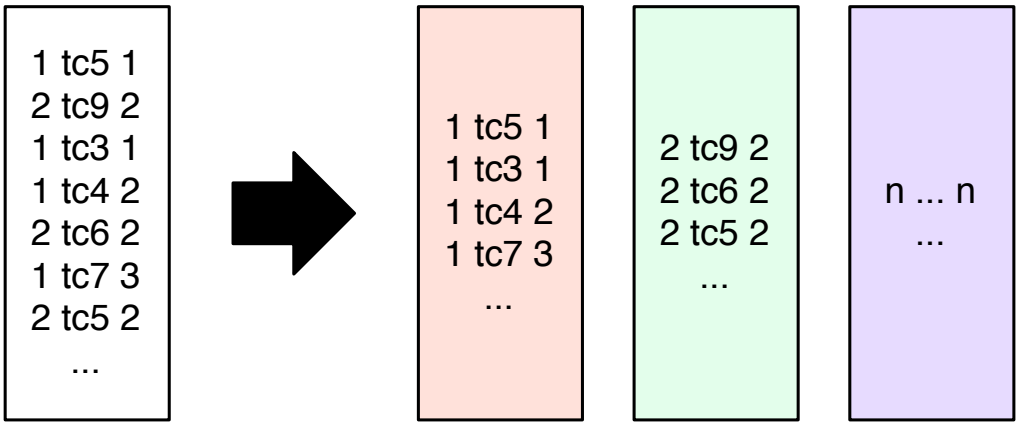

Figure 7.3: Split of Test Case Prioritization according to Preconditions

Precondition-based Grouping. To improve test efficiency for complex testing scenarios, we introduce a precondition based grouping of test cases to reduce the overhead between test case executions and form coherent scenarios. The original prioritized list is split into a subset of n lists, each containing only a set of test cases $TC_{pre} \in \mathcal{TC}_{sys}$ which are designed for the same precondition $pre \in \mathcal{PC}$. Besides a higher test efficiency due to reduced overhead, it also allows testers to parallelize the testing process by assigning different subsets of test cases to different testers while ensuring a certain coherence between the test sets due to their shared preconditions. To extend this approach, we also consider the difference between test cases which do not elicit a state change, i.e., precondition pre_k equals the postcondition $post_l$ for a test case $tc \in \mathcal{TC}_{sys}$, $k, l \in \{1, ..., |\mathcal{PC}|\}$.

Even though the split dissolves the stoic first prioritization into more coherent sublists, it raises another problem. Assume that the number of test cases for the different preconditions is very imbalanced, i.e., the subset of one precondition pre is much larger than another of a subset for pre'. Now, if the test set for precondition pre is executed before the other subsets, it may take a very long time to execute all test cases for this subset. As prioritization does not make any proposals regarding the selection of test cases, i.e., when to stop testing, resources may be consumed before any of the other precondition lists can be executed. Thus, a combination of both, *prioritization* and *selection* is desired.

To achieve this, we propose that an a priori number of test cases $sel \in \mathbb{N}$ is to be defined, for which the precondition split is performed. Hence, we take the list of the first sel test cases, named $TC_{selected} \subseteq TC_{ordered}$ and create a separate ordered test case list for each of the preconditions $pre_{selected} \subseteq \mathcal{PC}$ occurring in $TC_{selected}$, i.e., $|pre_{selected}| \leq |TC_{selected}|$. This allows to use the sel most important test cases for testing, while recognizing the preconditions for testing. However, this also forces

the testers to execute all of these test cases, as test cases of the last precondition might be the most important ones.

If there are no distinct preconditions, or the test cases are executed automatically with a low overhead, the precondition split has not to be performed.

Example 7.5: Splitting Preconditions

Assume the example shown on the left hand side in Figure 7.3 as output of our test case prioritization technique. We receive n lists after applying the precondition split, which is symbolically shown on the right hand side. For each precondition, an individual list has been created keeping the relative prioritization of each test case, i.e., test case *tc5* is still more important than *tc3*. It is possible to execute test cases for one precondition, still starting with the most important one and working forwards, e.g., with test case *tc5* for precondition 1 and *tc9* for precondition 2. This allows for parallelism between testers, when separately focusing on different precondition lists.

7.3 Evaluation

We evaluated our test case prioritization technique to show its effectiveness. First, we formulate three research questions which we answer in this evaluation. Second, we explain the subject systems for which we analyzed our test case prioritization approach. Next, we explain our methodology and present and discuss results afterwards. Finally, threats to validity are explained.

7.3.1 Research Questions

We formulate the following three research questions to assess the quality of our test case prioritization approach:

RQ1: *How* effective *is our test case prioritization approach and does the analysis of natural language artifacts increase effectiveness?* We assess the effectiveness of our test case prioritization technique in terms of its failure finding rate, i.e., how fast are failures in the system revealed using our prioritized list? In addition, we assess if the natural language related features improve the effectiveness compared to meta-data features. We compare our test case prioritization approach to a random prioritization.

RQ2: *How effective is our black-box test case prioritization approach in comparison to a test expert in real-life scenarios?* We compare our test case prioritization

technique in a real-life testing scenarios against a test expert in terms of failure finding rate. We also analyze the suitability of our results for different stopping criteria.

RQ3: *How* efficient *is our test case prioritization approach?* We analyze if our test case prioritization technique is applicable in real-life scenarios in terms of computation time.

7.3.2 Subject Systems

Similar to our multi-objective test case selection approach for black-box testing of software versions, we apply our test case prioritization approach to two subject systems: BCS and a real-life industry case study (cf. Chapter 3.2 and 6.3.2).

BCS. For our evaluation, we use the system-level artifacts of BCS as described in Chapter 3.2.2. To support the supervised learning, we split the set of 128 in two equal subsets, i.e., 64 test case have been defined as positive and the other 64 test case are used as negative for learning. In total, 8 failures have been seeded for these test cases as described in Chapter 6.3.2.

Industrial Data. A data set of real-life test cases, requirements and revealed failures has been provided by an industrial partner from the automotive industry (cf. Chapter 6.3.2). For our evaluation, we use a subset of 645 test cases, for which we are able to provide sophisticated labels for supervised learning with a high confidence. These test cases revealed a total of 34 failures. For training, we split the data into 354 positive and 291 negative test cases, using expert knowledge. This split has been performed by the authors, focusing on the most important parts of the system. Due to technical limitations, we have no information about the specific execution runs which led to the failures. This does not hamper with the test case prioritization, as we still have meta-data available.

7.3.3 Methodology

Implementation. We implemented our test case prioritization technique using the DLIB [Kin09] machine learning library. For supervised learning, we use the ranked support vector machine (SVM Rank) by Joachims [Joa02], which is able to handle large features spaces with sparse data. Test artifacts are stored in HP QUALITY CENTER (QC). To perform NLP-specific tasks, we use the LUCENE[1] library. The developed tool is not open source and is not publicly available due to legal restrictions.

[1] LUCENE text search engine library by APACHE FOUNDATION, url: `https://lucene.apache.org/core/`, date: March 30th 2017

Quality Assessment. To measure the effectiveness of our test case prioritization approach, we measure the *Average Percentage of Faults Detected* (APFD) metric [RUCH01] (cf. Chapter 2.2.2). The APFD function returns a value between 0 and 1. The higher the value, the higher is the quality of the prioritization. Other forms of APFD exist, such as *APFDc* [EMR01], which incorporate costs instead of test case positions for the quality assessment. However, the execution time (or cost) of each test case is a very subjective manner, e.g., depending on the tester which executes the particular test case. Thus, we use the position of test cases to evaluate the effectiveness of our test case prioritization approach. We do not consider test case preconditions for these orderings, as the APFD computation is performed for one test case list and we do not provide guidelines for a *good* amount of test cases selected for the precondition split (cf. Chapter 7.2.3).

Evaluating Attribute Combinations. For the evaluation of the test case prioritization, it is of interest to measure how each available feature influences its effectiveness. To this end, we execute the prioritization technique in multiple runs using a different combination each time. In detail, we combine the following six different attributes: *test case description* (TCD), *execution costs* (EC), *requirements coverage* (RC), *failure history* (FH), *failure age* (FA) and *failure severity* (FS). In total, we perform $2^6 - 1 = 63$ runs for these combinations. For BCS, the execution time of test cases is unknown. Hence, we only execute $2^5 - 1 = 31$ different combinations for BCS.

K-Fold Cross Validation. We evaluate our first research question on a snapshot of data, i.e., we do not perform an actual test run but use historic data collected in previous executions. To improve our evaluation and gain confidence about our results, we perform *k-fold cross validation* on the data [WFH11]. Cross validation allows to perform a more sophisticated analysis on smaller data sets and is generally used to assess the prediction error in machine learning [Fus09]. It is applicable in case of this thesis, as we only use already existing knowledge to compute the ranked classification model, which does not require knowledge of future failure detections. Also, our test cases do not depend on each other.

In k-fold cross validation, the data is split in k folds, using $k-1$ folds for training and one fold for validation. This is repeated k times, such that each fold is used once as validation data set. The repetitions are independent of each other. For k-fold cross validation, a suitable number of k folds has to be defined. For our industrial data set, we perform $k = 10$ folds, which is a standard value [Fus09]. As the BCS data set is about 80% smaller than the industrial data set we set $k = 5$ for BCS to avoid folds of small size. We perform the k-fold cross validation for all of the different attribute combinations, i.e., a total of $63 \cdot k$ (respectively only $31 \cdot k$ for BCS) folds are computed.

Preparing Failure Information. As we use failure history in certain attributes, but also regard the revealed failures for quality assessment, we need to perform a split between failures which are used for learning and for those which are used for validation. In particular, we decided to distribute failures according to their age into these two sets, i.e., we computed the average age of failures. Those older than average are used for learning (i.e., they are already known to the approach) and newer failures are unknown to the learning, but are used for APFD computation. This avoids any bias regarding the a priori knowledge of the technique. If no failure related attribute is used, no split is performed and all five failures are considered for APFD computation. In reality, new failures are discovered while testing, but as we used already existing data without explicit knowledge about execution runs, this artificial failure categorization has to be performed. For our second research question, we used actual failure information of a live executed regression test and did not perform a split.

Example 7.6: Failure Splits for Evaluation

Assume a total of 20 test cases, of which 5 are linked to revealed failures. We want to split the failures into two sets, one for learning (already known) and one for validation (new failures). To this end, we compute the average failure age and split into newer and older test cases, e.g., in this example three test cases could be older and the remaining two newer than the average date. Then, we use the three older failures as attributes when using failure related attributes, e.g., test case history. The other two test cases then are only used to validate the results, i.e., measure the APFD of the ordering.

7.3.4 Results

We evaluated our test case prioritization approach using BCS and the real-life industry data to answer our three research questions. We show and discuss the acquired results in the following.

RQ1: *How* effective *is our test case prioritization approach and do natural language artifacts increase effectiveness?*

BCS. We measure APFD for BCS performing $k = 5$ folds for a total of 31 combinations of attributes. The results are shown as boxplot in Figure 7.4.

The figure shows three different boxplots representing average APFD results for different prioritization sequences, derived by different attribute combinations. We

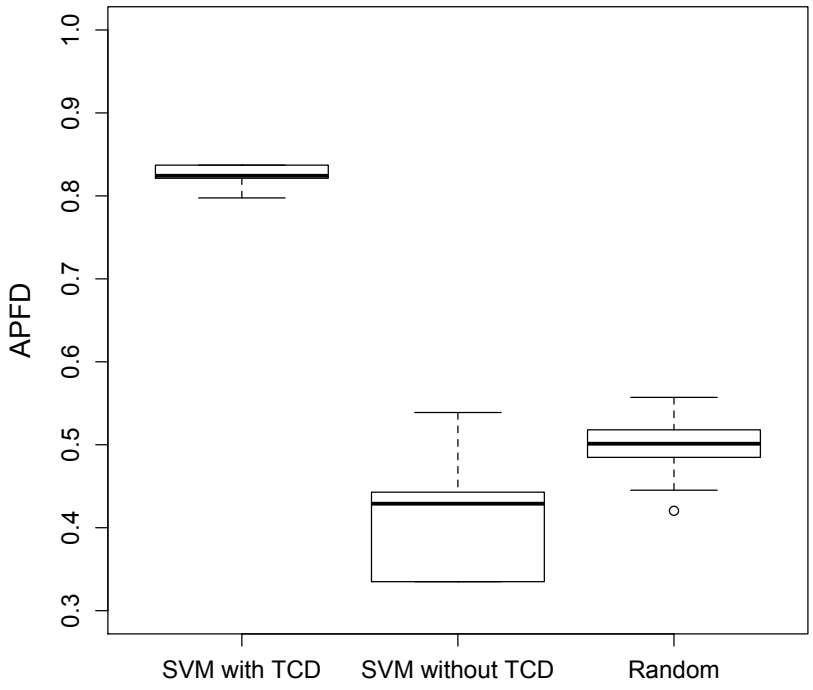

Figure 7.4: Machine Learning Test Case Prioritization Results for BCS

consider the six different attributes *TCD*, *FH*, *FA*, *FS*, *EC* and *RC* (cf. Chapter 7.3.3) and perform all 31 different combinations of them separately. That means, we perform the learning and prioritization using cross validation, using different features to be accessed by the ML. On the left-hand side of Figure 7.4, we show the aggregated results for the prioritization for all combinations including the test case description, i.e., all 15 combinations of attributes containing the TCD attribute. The second plot shows the data generated by all combinations containing attributes without the TCD attribute. For comparison, the third plot shows the results of 100 random prioritizations for each cross validation run. Both plots produced by our test case prioritization technique have been generated using SVM Rank [Joa02].

The measured metric indicates that the machine learning-based approach achieves good results for the BCS data. Given a median APFD of above 0.8 with a standard deviation (SD) $\sigma = 0.012$, it performs very well using TCD in addition to metadata (cf. left plot in Figure 7.4). The small σ-value shows, that the technique is very stable, no matter which other attribute combinations are used. The results are

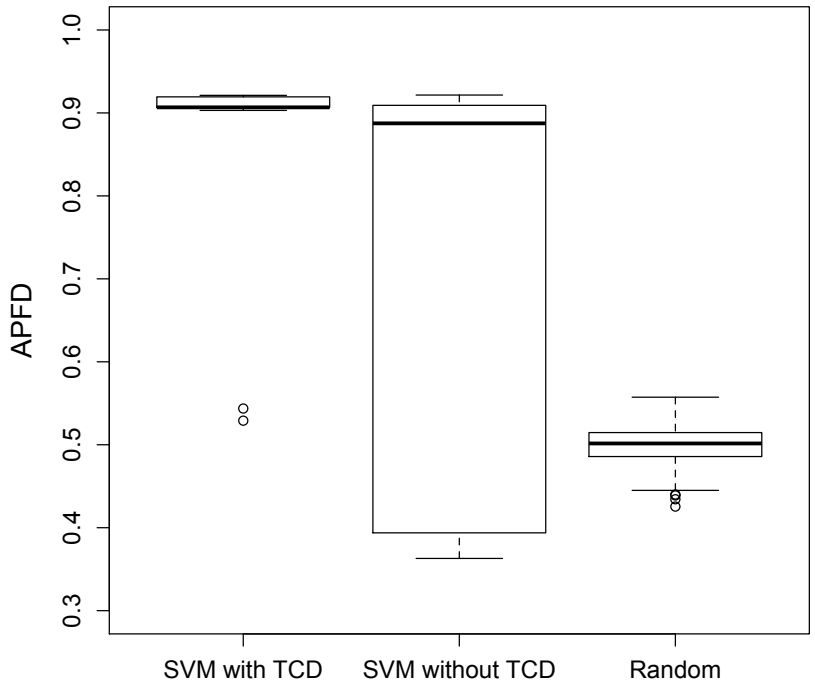

Figure 7.5: Machine Learning Test Case Prioritization Results for Industrial Data

very promising for all combinations using the TCD, especially when compared to the results achieved by the random approach, which results in a median APFD of 0.5 with $\sigma = 0.025$ after 100 repetitions (cf right plot in Figure 7.4). However, the machine learning technique has troubles to detect failures early without the TCD knowledge. This is shown in the second plot in Figure 7.4. These combinations did only lead to rather low APFD values, with a median of ~ 0.45 with $\sigma = 0.064$. Our evaluation shows, that the TCD attribute has a positive impact in the computation of the ranked classification model and, thus, should be selected when available. This validates our assumption that the test case description has a positive impact in terms of effectiveness.

Industrial Data. For industrial data, we use a set of existing data has been used to evaluate the effectiveness compared to a random approach. Figure 7.5 shows the APFD boxplots of different combinations of our test case prioritization technique and the random prioritization approach. We show the same three different result distributions, i.e., our test case prioritization approach with the test case description,

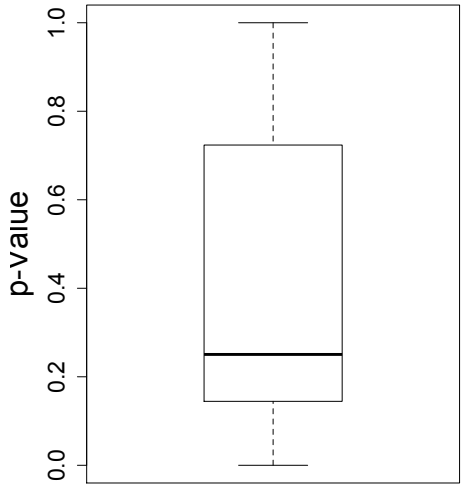

Figure 7.6: Distribution of P-Values for ML classifiers for Industry Data

without the TCD knowledge and the random distribution of 100 random runs for each ML run. It becomes evident, that our ML-based prioritization approach is able to outperform the random prioritization by far. Moreover, the usage of TCD as feature for the ML-based prioritization increases the APFD significantly, leading to an median APFD value of 0.9, with a SD $\sigma = 0.13$, which shows a rather stable data distribution. Without TCD knowledge, the median APFD is still at about 0.88, with a larger SD of $\sigma = 0.24$, with the lower quartile stretching down to an APFD value of 0.4. This supports our findings for BCS, i.e., the usage of the TCD feature is useful in prioritization to improve the APFD. The random prioritization was only able to achieve a median APFD of 0.5 with $\sigma = 0.022$.

We further assess the significance of the different distributions produced by the different objective combination. Using the *Mann-Whitney-U* significance test [MW47], we first analyze the statistical significance of the results produced by combinations with TCD vs. combinations without TCD. The resulting p-Value is 0.003, which is below our alpha-value of 0.05 and, thus, presents a significant difference in results. For a more detailed analysis, we also computed p-Values for all pairs of combinations for our black-box test case prioritization for software versions. This leads to a total of 1953 significance tests. For the sake of understandability, we present a boxplot showing the p-value distribution for all significance tests of all learned classifiers using all different objectives for industry data in Figure 7.6.

The plot indicates that, in fact, many distributions are not statistically significant. This is to be expected, as the most combinations are combined with a very similar amount of objectives. Thus, the underlying classifiers are similar.

Overall, we validate our second research question, showing that our test case prioritization approach is effective in terms of APFD compared to a random approach. Additionally, we are able to show that the test case description features are able to improve the results further.

RQ2: *How effective is our black-box test case prioritization approach in comparison to a test expert in real-life scenarios?*

For our second research question, we compare the machine learning-based prioritization to test case executions orders of actual test experts, using new and previously unknown failures. A setup overview of the experiment is shown in Figure 7.7. To guarantee a fair comparison, the same expert we compared our test case prioritization technique to also trained the system. In fact, the same test expert trained the system twice, generating two different classification models, to which we refer as *Classifier A* and *Classifier B*. Classifier A has been created without any previous knowledge about the ML-based prioritization technique or ML in general, using 236 positive and 158 negative test cases. They belong to the same SUT as in the first experiment on static data, but are now executed in a live scenario, revealing new failures. The second classifier, Classifier B, was created after the results of Classifier A were known to the tester. He used more time to identify important test cases for the second classifier, which led to 255 positive and 155 negative test cases for training.

In general, the test expert reported that negative test cases were harder to select as testers are mainly interested in useful test cases. Hence, the negative test set is smaller than the positively labeled test cases. The negative set contains test cases, which have not been executed for a long time, or are not relevant for the current system, e.g., as their functionality is not present anymore.

We apply SVM Rank to learn one ranked classification model for each set, using all available information, including TCD. The resulting models then have been used to rank a total of 155 test cases, for which it was known that they are executed. As the test expert does not know, which test cases actually reveal failures, the manual ordering is based on intuition and expert knowledge. Due to the large quantity of available test cases, the testers do not perform an a priori prioritization of test cases, but order them in an ad-hoc fashion while executing them.

After the prioritization has been computed, the test cases have been executed by the expert without access to the automatically generated list. After the execution had been finished, we compared the execution order and APFD values of both, the

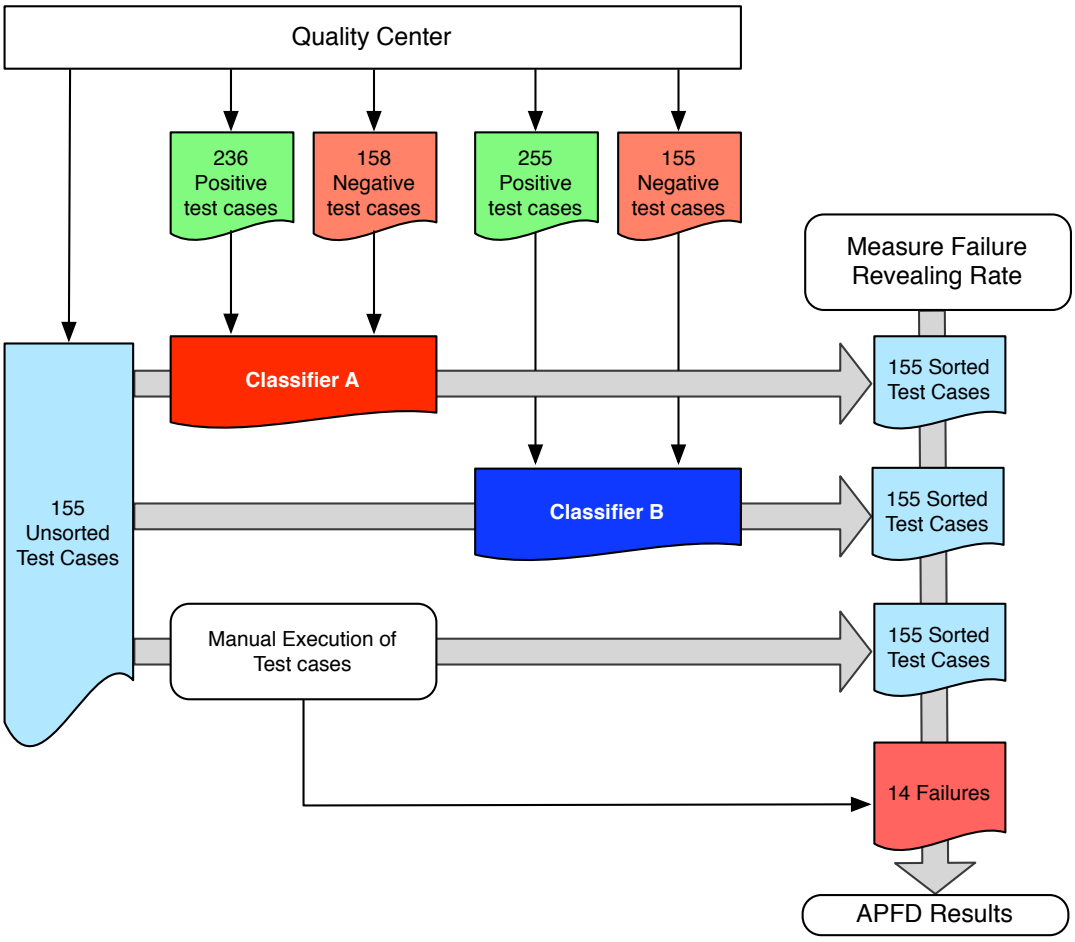

Figure 7.7: Experiment Setup of Comparison to Human Tester

prioritization derived by two different classifiers and the actual manual ordering of test cases. The results are shown in Figure 7.8 represented as colored lines. The orange line represents failure finding rate of test case order by human expert. It is compared to the two different classifiers, depicted in red (*Classifier A*) and blue (*Classifier B*). The average value is shown as black dotted line, it represents the expected value of a random approach as comparison. The number of executed test cases is represented as x-coordinate. If an executed test case reveals a failure, the y-value is increased by one, i.e., the ordinate shows the number of revealed failures.

In total, the 155 executed test cases revealed only 14 failures, i.e., only 10% of the test cases have been successful in revealing a failure in the system. The APFD values for the human expert's ordering is 0.53, while the APFD of for Classifier A is 0.60 and APFD for Classifier B is 0.62 and, thus, both automatically generated sequences are

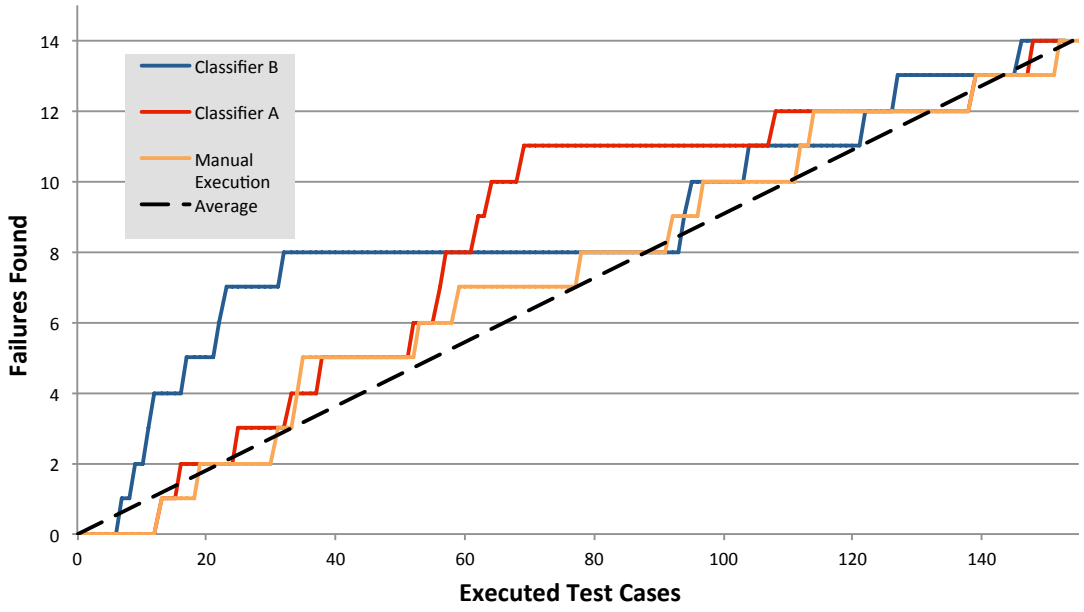

Figure 7.8: Comparison Human vs. Machine

significantly better than the manual execution order. Both classification techniques are also superior to average execution shown in the black line, which indicates that they work better than a random approach. Unfortunately, a drawback of our used ML technique is the problem to comprehend the learned models. Right now, a learned hyperplane is not comprehensible by a human expert as it is represented in an n-dimensional space. Thus, we are not able to resolve why one classifier is different from the other. However, we favor a good prioritization result compared to understanding what leads to a particular ordering for our work.

Evaluating Test End Criteria. The prioritization of test cases is useful, as in reality complete testing is not feasible due to the sheer complexity of modern SUTs [AO08]. Hence, a prioritized set of test cases is probably not executed until the last test cases, otherwise we would not need a prioritization at all. Hence, regression testing requires certain test end criteria, which define the number of test cases to be executed [LV04]. Thus, we give some insights how many failures are revealed by only executing certain percentages of all test cases. For evaluation, we compare the results for the test end criteria at 90%, 75%, 50%, 33%, 25% and 10% of all test cases executed. This gives some insight in the actual meaning of the APFD results shown in Figure 7.8, as it becomes more clear how the failure finding rate is actually influenced by our test case prioritization technique. The measured results

Table 7.1: Failure Findings at Potential Testing End Criteria

% TC	% Failures Class. A	% Failures Class. B	% Failures Expert
90%	**92.8%**	**92.8%**	**92.8%**
75%	**85.7%**	78.5%	**85.7%**
50%	**78.6%**	57.1%	57.1%
33%	35.7%	**57.1%**	35.7%
25%	35.7%	**57.1%**	35.7%
10%	14.3%	**28.5%**	7.1%

are shown in Table 7.1 showing the percentages of test cases executed compared to the percentage of failures found.

The highest percentages of revealed failures achieved for each end criterion are marked in bold font. It becomes evident, that both classifiers are strong when executing the first 50% of test cases. In particular, Classifier B is the strongest up to a third of all test cases executed, where it manages to capture 57.1% of all failures. Actually, this mark is already reached at about 20% of all test cases, which indicates a classifier that is very strong for few test cases executed, which would be optimal in short regression cycles. However, when executing 50% of all test cases, Classifier B is outperformed by Classifier A, as it still only detects 57.1% of all failures compared to the 78.6% of Classifier A. The manual order is outperformed by Classifier A at this point. When executing 75% or more of all test cases, the manually applied ordering is able to catch up to the machine learning techniques. But it is arguable if that many test cases are actually executable in regression testing scenarios. Hence, we claim that our test case prioritization technique is very effective when executing only 50% or less of all test cases.

Summarizing, we validate our second research question, showing the effectiveness of our test case prioritization technique compared to a test expert. We further claim that the resulting lists are suited to support testing with low resources, even when stopping after $\sim 30 - 50\%$ of test cases.

RQ3: *How* efficient *is our test case prioritization approach?*

In terms of efficiency, we measure the execution times of our black-box test case prioritization approach. We distinguish between the training duration and test case classification times. The average execution times measured for BCS and industry data are shown in Table 7.2.

Table 7.2: Training and Classification Durations of the Prioritization

	BCS	Industry
Training Duration		
Average of Combinations with Test Case Description	47.1 ms	2,900 ms
Average of Combinations without Test Case Description	8.6 ms	127 ms
Classification Duration		
Average of Combinations with Test Case Description	5.4 ms	134.7 ms
Average of Combinations without Test Case Description	0.6 ms	8.7 ms

In general, training takes a lot longer than classification, which is due to the fact that building a ranked classification model from scratch is much more complex than classifying test cases according to an existing model. As the number of features is an important influence factor for the complexity of models, we also separate the measured times between combinations with test case description and without, as the usage of the description increases the feature dimension space significantly. For BCS, the average training time is between 8.6 ms without and 47.1 ms with test case description features. With a similar ratio, the training duration for industrial data is between 127 ms and 2,900 ms. Even though the learning takes about up to 60 times longer than for BCS, the computation times are still negligible in realistic scenarios, as the classification takes only mere seconds using a total of 645 test cases and, thus, does not negatively effect current regression testing processes.

Classification of test cases takes mere milliseconds for both systems. The longest measured classification duration takes 134.7 ms for industry data. This includes the test case description features as well as meta-data-related features.

Consequently, we validate our third research question, arguing that a test case prioritization can be performed on a daily basis without noticeably delaying the testing process for even larger data sets.

7.3.5 Threats to Validity

We split the investigation of threats to validity into internal and external quality according to Runeson and Höst [RH09].

Internal Threats to Validity. Different aspects influence the outcome and quality of our test case prioritization approach. First and foremost, test data should be prepared before learning commences [WFH11]. In particular, we noticed that there is a lack of sufficient traceability between the different artifacts, especially between test cases and their revealed failures. One reason for this is *exploratory*

testing of systems, e.g., testing a driver assistance system in the car in realistic scenarios, where test cases are not explicitly executed, but failures are found ad-hoc by exploring the system functionality or performing use cases [IR05]. The lack of traceability reduces the amount of data available for our black-box test case prioritization and restricts the ability to learn meaningful models. Another important aspect is the quality of the provided artifacts, i.e., the correctness and actuality of the meta-data, requirements, failures and test case descriptions. For the evaluation, we made sure to use test cases which are linked to requirements by experts. In addition, failures have also been linked, when revealed. The descriptions have been created by experts as well, even though they can differ in certain aspects corresponding to their author.

The quality of selected training data influences the learned model as well. Different training data sets might lead to better or worse solutions and, thus, it needs to be investigated in more detail how to derive good training data. In our evaluation, we present two different training data sets in our comparison to a test expert, both created by the same expert. The results show, that the quality of the prioritization depends on the selected data and can be steered into a certain direction. However, due to the complexity of the underlying ML algorithms and their learned models, it is difficult to grasp the actual differences in the learned models. This makes assumptions about good or bad training data non-trivial. We propose to perform more case studies, which focus on analyzing the given training data and their resulting models. Furthermore, models computed by other techniques might be easier to interpret. For instance, *decision trees* contain the rules learned from the training data [SL91]. The applicability of these techniques has to be evaluated in future work.

External Threats to Validity. An external threat to validity is the restricted number of two case studies, for which our black-box test case prioritization approach has been performed and evaluated. For different systems and test experts, the results might differ, as test case descriptions and expert decisions are of subjective nature. Each tester decides differently on how to select training data, according to their knowledge and role in the project. Hence, additional case studies are required to deduce an overall picture on how effective our test case prioritization is, especially in different scenarios. To counter this threat, we performed an academic and industrial case study with real-life data. Furthermore, we integrated our test case prioritization technique in a live testing process to compare the results to a test expert. Hence, we are able to present a first indication of the applicability of our test case prioritization approach with promising results.

7.4 Related Work

Mainly, two directions of research are investigated related to our test case prioritization approach. The first direction are *black-box regression testing techniques*, which currently do not apply machine learning and are often limited in terms of their used input artifacts. The second domain of interest are *machine learning based regression testing techniques*, which, as of date of this thesis, heavily focus on white-box testing. An overview of all presented related work is given in Table 7.3. For each mentioned related work, we show a categorization of the technique in terms of prioritization and machine-learning based as well as the used artifacts. In addition to artifacts used in by our approach for black-box test case prioritization, we list *risk*, *code* and *other*, to give a more comprehensive overview, especially as some related techniques use more than one type of artifact or use different types of artifacts not available for testing in case of this thesis, while they are still worth mentioning due to similar ideas or concepts. **Henard et al.** [HPH+16] performed an extensive case study on the comparison of white-box and black-box prioritization techniques. They state, that white-box is still the more thoroughly investigated area, but their comparison shows that black-box and white-box are very similar in terms of fault detection rates. However, in terms of black-box testing, they only analyze model-based or input/output-based techniques, which are not available in case of our test case prioritization approach.

Black-Box Regression Testing. In regression testing, a large number of techniques have been derived in the past, but only a fraction of existing work focuses on the absence of code or model information or system level testing [YH07b, ERS10]. We present the most related work, which actually focuses on black-box data similar to the data used in this thesis. **Chen and Wang** [CW14] introduce a test case prioritization technique for specification-based environments, based on requirements severity. They introduce a novel requirements severity scale and also examine dependencies between test cases, which they formulated as sequence ordering problem. For realization, they use genetic algorithms and ant colonization algorithms, which show similar results in a synthetic case study. In contrast to our test case prioritization approach, they do not analyze other artifacts and do not analyze test case descriptions. **Fazlalizadeh et al.** [FKAP09] introduce a test case prioritization technique which incorporates history data, such as failure history of test cases, to compute priority values without source code knowledge. They apply a greedy test case prioritization strategy without machine learning. Their work has been extended by **Engström et al.** [ERL11] by using new information sources such as a static priority value or the test case creation date. However, Engström et al. [ERL11] noticed no considerable improvement compared to the approach by Fazlalizadeh et al. [FKAP09], but they do show that the results of the greedy prioritization improve the ability of a test

Table 7.3: Categorization of Related work for Black-Box Test Case Prioritization

	Technique			Used Artifacts					
Related Work	**Prioritization**	**ML**	**TCD**	**REQ**	**FH**	**Costs**	**Risk**	**Code**	**Other**
Black-Box Regression Testing									
Chen and Wang. [CW14]	X			X			X		
Fazlalizadeh et al. [FKAP09]	X				X				
Engström et al. [ERL11]	X				X				X
Hemmati et al. [HFM15]	X		(X)				X		X
Huang et al. [HPH12]	X				X	X			
Ledru et al. [LPBM12]	X		X						
Noguchi et al. [NWF+15]	X				X				X
Qu et al. [QNXZ07]	X				X				X
Srikanth et al. [SBWO14]	X			X	X		X		X
Yu and Lau [YL12]	X				(X)				X
Machine Learning in White-Box Testing									
Gondra [Gon08]	X	X						X	
Jiang et al. [JCM07]	X	X		X				X	
Machine Learning in Black-Box Testing									
Agarwal et al. [ATLK12]		X			X				X
Bhasin and Khanna [BK14]	X	X						(X)	X
Briand et al. [BLB08]		X							X
Perini et al. [PSA13]	(X)	X					X		X
Yoo et al. [YHTS09]	X	X							X
Our Test Case Prioritization Technique	X	X	X	X	X	X			

Prioritization = Test case prioritization technique, **ML** = Machine Learning-based approach, **TCD** = Test case description, **REQ** = Requirements-based approach, **FH** = Failure history

suite to detect faults early and to improve transparency for testers. **Hemmati et al.** [HFM15] prioritize manual test cases, comparing three different heuristics: *covering maximum topics*, *test case similarity* and *risk-based clustering*. Similar to our test case prioritization technique, they extract a predefined number of topics out of textual test case descriptions. However, they aim to cover many different topics as fast as possible. In their comparison, they notice that risk-based heuristics yield the best results for the compared industrial data. For traditionally developed software, they indicate that results are not much more effective than a random prioritization. While their techniques also aim to support manual black-box testing, they use different concepts and artifacts compared than our test case prioritization technique. **Huang et al.** [HPH12] describe a cost-cognizant test case prioritization for black-box test cases, measuring APFDc [EMR01]. Their prioritization is based on fault information, test case costs and fault severity using a genetic algorithm. They do not analyze test case descriptions or other meta-data. **Ledru et al.** [LPBM12] present a prioritization technique, which is solely based on test case descriptions. They restrict their technique to the input of test cases. They use a greedy algorithm to order the test cases according to the distances of their string representations. For evaluation, they assess the APFD of four different distance metrics and a random prioritization. They find that the best results are achieved using the *Manhattan distance*, which also outperforms the random prioritization on average. **Noguchi et al.** [NWF+15] apply ant colony optimization to prioritize test cases for black-box data. In their approach, test cases are manually grouped into test categories which are enriched with precedence constraints between different categories. In addition, historical data from previous products is required. Based on this, their technique is able to create a prioritized category list. **Qu et al.** [QNXZ07] present a test case prioritization for black-box systems without source code knowledge based on test results and run-time data. They use a matrix which contains test case relations, such that higher prioritized test cases can influence the priority of related test cases. This is similar to our theory of the categorization of test cases via the machine learning classification model. In contrast to our work, Qu et al. build the relationship matrix based on the fault history of the test cases, i.e., similar fault types revealed mean that test cases are similar. Furthermore, they assume that each test case can only detect a certain type of regression fault. **Srikanth et al.** [SBWO14] present the PORT prioritization technique. Their approach uses requirements as input and performs a complex analysis on different aspects, i.e., *customer-assigned priority*, *requirements complexity*, *requirements volatility* and *fault proneness*. The authors find that more complex requirements result in a higher number of faults and that the approach overall is able to improve the failure detection rate compared to random ordering. Similar to our test case prioritization approach, they prioritize system test cases on black-box data. In contrast to this thesis, they use a greedy

technique which solely focuses on requirements in combination with linked faults for the priority computation. In addition, they assume to have knowledge about *customer-assigned* priorities for requirements, which have a significant influence on the prioritization quality. This shows, that expert knowledge can positively influence the output of a automated prioritization and should be considered for regression testing. **Yu and Lau** [YL12] present a fault-based prioritization technique based on *fault models*. It is applicable without code information and requires test cases generated using different fault models, i.e., it is known which faults are likely to be detected by the test cases and there are known relationships between the faults. They are able to reduce the testing effort compared to other techniques.

None of the above techniques apply any form of machine learning techniques or try to emulate decisions made by the testers or incorporate the test case descriptions to compute ranked classification models.

Machine Learning in White-Box Testing. In white-box testing, machine learning based techniques are often described using the term "software fault prediction". Here, learning is performed on code-level, i.e., features are extracted directly from the source code and its modifications. Surveys show, that different fault prediction techniques and many corresponding case studies are already explored in the literature [CD09, Cat11]. Due to the white-box nature of these techniques, we only name a few approaches which have been investigated in recent years and are close to this work as they include certain aspects used in this thesis. **Gondra** [Gon08] applies artificial neural networks (ANN) and SVMs to classify if a software module is fault-prone and compares the results of both techniques. As a result, SVMs outperformed ANNs in the classification task. Many of the used metrics are of white-box nature, e.g., LOC, McCabe cyclomatic complexity or Halsteads Program Length. His work shows the potentials of SVM-related techniques in testing tasks, which is similar to the observations made in this thesis. **Jiang et al.** [JCM07] present a fault prediction technique based on early life-cycle artifacts such as requirements descriptions and code metrics. They compare different machine learning techniques in their work. A performed evaluation indicates that only requirements-based metrics did not result in good fault prediction, but when combined with code metrics could enhance the overall prediction quality. This shows future potential for our test case prioritization technique when combined with code-level information.

Machine Learning in Black-Box Testing. Machine learning has also been sparsely applied to certain black-box testing problems. **Agarwal et al.** [ATLK12] compared two different machine learning techniques, ANNs and info-fuzzy networks, in their ability to be applied as automated oracles in black-box testing. They find that the results of the techniques heavily depend on the amount of available data. In contrast to this thesis, these techniques are used to determine if a test case has revealed a fault or not based on program inputs and outputs. **Bhasin and**

Khanna [BK14] propose a black-box testing technique based on neural networks. They assume that test cases are represented as module state diagrams, which represents functions and their calling sequences in a graph-like fashion. These diagrams are either retrieved from stack traces or the software specification. They identify and prioritize module interactions and use them as input for the neural network. In contrast to our test case prioritization approach, they require a certain representation of module interactions and have to prioritize them a priori to train the neural network. **Briand et al.** [BLB08] introduce the *MELBA* regression testing technique for test suite refinement in black-box testing. It is based on *C4.5 decisions trees* [Qui93] and the category-partitioning method [OB88]. Test cases are abstracted to make propositions about the test suite quality and the need of possible re-engineering tasks to cope with test suite weaknesses. Their technique is semi-automatic and uses test case abstractions. Hence, it does not make use of test case descriptions or meta-data per se, but representations created using in and output information of test cases. **Perini et al.** [PSA13] present the case-based ranking (*CBRank*) method to prioritize requirements instead of test cases. It uses information by the project's stakeholders and priority values learned by a machine learning approach. They apply a boosting method using different classifiers and ranking functions. The presented approach neither considers natural language or meta-data information nor the prioritization of test cases, but only requirements. **Yoo et al.** [YHTS09] present a clustering-based test case prioritization. The clusters are computed based on the dynamic execution traces of the test cases, which are not available in case of this thesis. Afterwards, the computed clusters are prioritized by a human test expert to make use of the domain knowledge. Hence, the technique differers greatly from our supervised technique that automatically prioritizes test cases based on their black-box data, presented in this thesis. All of these techniques do not consider natural language artifacts and meta-data as input for a test case prioritization.

Looking at the related work presented in Table 7.3, we observe that their exists only few work regarding the application of machine learning in black-box testing. In particular, no other technique uses all the data analyzed by our test case prioritization approach, especially the test case description is rarely used and never in the context of supervised learning. In contrast, the failure history is regarded in eight different approaches. A total of ten presented techniques uses different artifacts, which have not been available in the context of this thesis. This shows that our technique is a novel approach for the test case prioritization problem for black-box testing of software versions.

7.5 Chapter Summary and Future Work

Conclusion. We have shown that black-box test case prioritization is a difficult problem, which is currently not sufficiently investigated in software engineering. To this end, we presented a novel test case prioritization technique, which improves regression testing in an automated fashion. It is based on supervised ranked classification techniques [WFH11, Joa02]. The input is a set of positively and negatively labeled test cases. Test cases are contain natural language descriptions and black-box meta-data, such as test case execution costs. As the technique is trained by experts, it is highly adaptable to different domains or projects. For our evaluation, we used two subject systems: BCS and an automotive industrial data set. Our results show that our test case prioritization approach achieves very promising results in terms of the failure finding rate, especially compared to random techniques. Furthermore, we compared our test case prioritization technique in a live test run against a test expert. Again, the machine learning-based prioritization was able to increase the failure detection rate for different training data sets. Our evaluation shows, that our test case prioritization approach is efficient in its execution. This shows the potential of our test case prioritization technique to improve the difficult task of manual testing in real-life, large scale software development projects.

Future Work. For future work, several directions of research should be pursued. First, different machine learning techniques should be applied to the black-box test case prioritization problem, to assess if there is more potential using other algorithms to further improve the prioritization quality. An example are artificial neural networks, as investigated by related work in white-box testing (cf. Chapter 7.4). These types of algorithms have gained a lot of awareness in recent years, one prominent example being the *AlphaGo* algorithm implemented by Google DeepMind to beat human champions in the board game *Go* [SHM+16]. This system used deep neural networks in combination with other techniques, such as reinforcement learning, which led to an unprecedented success of an artificial intelligence in a very complex field, beating human experts. Hence, similar ML algorithms could be applied to solve the test case prioritization problem and incorporate additional data, e.g., knowledge from different projects.

Another type of machine learning concept that should be investigated in the context of this topic is *reinforcement learning* [SB98]. It allows to adjust the learning mechanism after a model has been learned by collecting feedback about the prioritization quality. As reinforcement learning requires a policy to follow, it has be to investigated on how these ideas can be transformed to test case prioritization. As a result, the resulting test case ordering might be influenced in a more fine-granular fashion. This could also reduce the effort of adapting training data for ongoing regression testing.

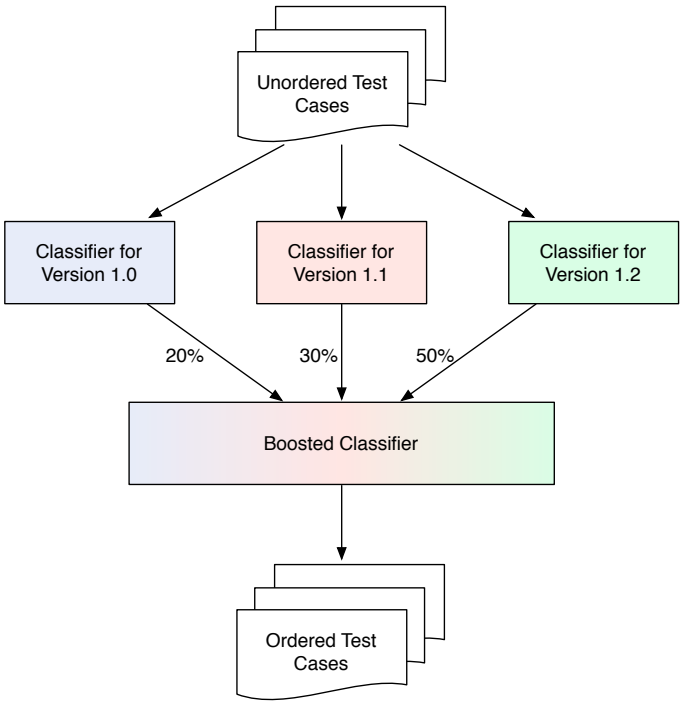

Figure 7.9: Boosting Concept for Test Case Prioritization

Second, instead of using one classification model at the time, which is replaced each time a new learning process is completed, previous classification models can be reused by applying a boosting technique [FS97]. A schematic structure for such a boosting approach for test case prioritization is shown in Figure 7.9. In the example, three different classification models are available. They have been created for different versions of the software, i.e., version 1.0, 1.1 and 1.2. Assume that version 1.2 is the newest version. Hence, the corresponding classification model should have the highest impact on the final classification, as it represents the current status of the project. However, the older models shall be regarded as well for test case prioritization as they have been valid in the past and might have been very successful in their produced failure finding rates. We propose that the older the classification model is, the lower should its impact on the final prioritization be. A boosting approach could increase the stability of results and introduce the possibilities to reuse other models which have been learned either in earlier project phases or, e.g., in similar other projects in the past.

Third, the case study diversity has to be increased, as the generalizability of the results is currently not given and, thus, the technique should be applied to other real-life data sets. In particular, more comparisons to human experts spanning longer time frames are required to assess the quality of the prioritization. Guidelines for

good training data selection for this particular testing domain have to be established and mined from future experiences made with the test case prioritization approach to improve the overall process. Additionally, the split of the preconditions needs more investigation. However, a first industrial case study has shown promising results on both, a static legacy data set and compared to human decisions made while testing black-box software versions.

8 Conclusion

Testing is an important task in modern software engineering [AO08]. However, testing a large number of product variants or rapidly developed new software versions is challenging. Testing resources are often not sufficient to retest every aspect of each product variant or version again. Thus, regression testing techniques have been proposed to either select or prioritize important test cases [YH07b]. However, as we have motivated in the introduction of this thesis, there is a lack of black-box regression testing techniques for both, software variants and versions. Following, we gave necessary background to understand the concepts and challenges tackled in this thesis. Next, we described the foundations of our testing techniques for software variants and versions, i.e., which artifacts we use to improve testing and how they are formally defined. Using both, the background and foundations, we are able to present our contributions.

In particular, this thesis contributes two approaches for testing of software variants in Part 2 and two approaches for testing of software versions in Part 3. We use the Body Comfort System [LLLS12] case study to evaluate all our testing approaches as well as an industrial system for testing of software versions. In Part 2, we have shown two novel techniques to improve testing of product variants in software product lines (SPL). Here, the main challenge is to test a potentially large number of product variants which are very similar, which leads to redundancy in testing. Retesting all applicable test cases for each product variant is infeasible for large SPLs. To this end, we proposed a new SPL testing framework, which helps to prioritize test cases for individual product variants under test [LLL$^+$15, LLAH$^+$16, LLS$^+$17]. Using a model-based [UL07] and delta-oriented [CHS10] approach, our test case prioritization technique for software variants is able to analyze changes between product variants to identify important parts of the system. We do not restrict our framework for particular types of test models. Based on changes between test models, we compute priority values for test cases for different product variants. Our evaluation has shown, that our testing framework is extensible, flexible and provides effective solutions to the test case prioritization problem.

Our second contribution in Part 2 introduces a technique to prioritize test cases for SPLs in a risk-based fashion [LBL$^+$17]. Currently, risk-based testing (RBT) is prominent in single-software systems, identifying important parts of the system [ELR$^+$14]. However, only one previous work has proposed an approach for RBT in SPLs [HvdLB14]. Compared to this technique, which focuses only on prod-

uct level risk assessments, our risk-based test case prioritization technique is able to semi-automatically derive risk values for test model elements in each variant under test. This reduces the effort compared to manual or single-software RBT approaches significantly. In particular, our risk-based test case prioritization technique is able automatically compute failure probability values. Our evaluation has shown the potential to outperform other techniques in an integration testing scenario.

Our first contribution in Part 3 improves black-box testing of software versions. To tackle the problem of regression testing without source code access, we propose a multi-objective test case selection technique. It identifies important test cases for each software version under test based on black-box meta-data [LFN+17]. While many test case selection techniques require white-box data [ERS10], we propose a black-box approach. To this end, we propose seven different black-box objectives to be optimized. The optimization is performed by search-based algorithms. In particular, we apply a genetic algorithm to solve the multi-objective problem. Consequently, we compute Pareto-optimal solutions, which dominate any other solution. Our evaluation shows, that the results are desirable with a precision similar or higher than retest-all, while reducing the test set size by 50% or more.

Our second contribution in Part 3 introduces a novel test case prioritization technique in regression testing of software versions. Again, the approach is solely based on black-box data. Instead of using source code, we use meta-data, such as test case history, as well as natural language descriptions of test cases to identify important test cases. This is done using a machine learning approach. We use ranked support vector machines [Joa02] to compute a ranked classification model for our data in a supervised fashion. Using the learned model, we are able to prioritize test cases defined in natural language. Our evaluation indicates, that our technique is able to outperform both, a random approach as well as the human expert in identifying failures early.

All of these techniques have been shown to be novel due to extensive related work analysis provided in the corresponding chapters.

8.1 Discussion

Testing of Software Variants. Our contributions for testing of software variants support the testing of individual product variants. We use model-based techniques inspired by regression testing to prioritize test cases. While our SPL testing technique is based on delta-oriented test models, the availability of test models in industrial context is rare. Usually testing is performed either manually or automatically without the usage of test models. This requires manual test case definitions. As model-based testing has numerous advantages (e.g., automatic test case generation, easy to understand specification, etc.) [UL07], we do not consider a model-based ap-

proach as a drawback. Instead, we propose that our technique for testing of software variants shows the advantages of model-based testing especially when dealing with large product spaces. If no test models are available, our test case prioritization technique cannot be applied.

Similarly, we use delta knowledge to identify important changes between product variants. While delta-oriented modeling [CHS10] and delta-oriented testing [LLL+14] have been proposed in the past, they specifically require delta knowledge, which is seldomly given in current SPL engineering processes. However, we argue that deltas do not need to be explicitly modeled, but can be extracted from existing SPLs, e.g., by using model-based delta generation [WRSS17]. In addition, we also consider other SPL modeling techniques as viable option for our testing approaches. While no availability of delta knowledge requires an adaption of our presented techniques, our framework and RBT technique are able to support different aspects of testing due to their generic nature. For example, differences could be analyzed using model mining techniques [WSS16]. This might increase the computation time to identify important changes compared to a delta-oriented analysis. Thus, an evaluation analyzing potential trade-offs has to be performed in the future. Consequently, we do not see a restriction to delta-oriented techniques, but see an applicability to different SPL concepts as well.

In terms of SPL testing, the gathered evaluation results show an improvement in the effectiveness in terms of the change covering rate compared to other techniques. However, our results are based on only case study, namely BCS. Further case studies have to be performed to generalize our findings. Realistic failure data is necessary to investigate how the technique performs in industrial case studies in terms of failure finding rates. This is a challenge as large SPLs are usually not open source.

Testing of Software Versions. In the third part of thesis, we contributed new approaches for black-box testing of software versions. The first software version related contribution uses a multi-objective optimization to identify important test cases for test case selection. Here, we define optimization objectives which are available in system testing and do not require source code access. Unfortunately, we were not able to execute all seven objectives and all of their combinations for both systems, nor could we execute all seven in combination as both case studies did only provide different subsets of the required meta-data. One particular open question is the influence of risk-based objectives when selecting test cases for real-life data. While we were unable to evaluate these objectives for our industrial data set, we argue that risk-based information is very beneficial especially in safety-critical systems, where certain standards have to be fulfilled (e.g., ISO 26262 [ISO09]). Here, risk-based data can help to identify the most important test cases according to defined risk values. The applicability and effectiveness of our test case selection technique to real-life risk values has to be shown in future work.

Looking at the results of our test case selection technique, we argue that a combination of black-box and white-box optimization objectives should be investigated in the future to further improve results, especially for real-life systems. Using solely meta-data restricts the effectiveness of our test case selection approach, as meta-data does not necessarily reflect changes between versions. However, code changes are a main source for failure between versions and, thus, further information should be incorporated to improve precision and recall of our test case selection technique. Due to the generic nature of our test case selection approach and the use of genetic algorithms, new objectives can be easily integrated. Previous work has already exploited multi-objective test case selection including white-box objectives [YH07a]. Thus, a combination of existing work and our test case selection technique should be examined in the future. Here, the challenge lies in the connection of data, i.e., traceability between source code fragments, requirements and test cases is required.

Our second contribution for black-box testing of software versions is our test case prioritization approach using machine learning. Here, our concept includes the analysis of natural language test cases. The approach has shown very good results in terms of APFD, i.e., it was able to outperform both random and manual test case prioritization approaches. In fact, the ability to outperform the human test expert shows the potential of our test case prioritization approach to be a guideline for testers in industrial settings, where the number of test cases is large and a manual analysis for each software version is infeasible. One drawback of our test case prioritization approach is the black-box nature of the learned classifier. Using SVM Rank [Joa02], we produce a mathematical model of immense complexity, which cannot be understood by test experts. This makes the integration of our test case prioritization technique difficult, as the test experts do not know why the machine is prioritizing certain test cases over others, only that it does. This is a problem with many modern machine learning approaches, such as deep learning [LBH15], where the learned models or rules are too complex and abstract to be intuitively comprehensible. Furthermore, the origin of certain rules may not be traceable either for a human expert. This reduces the confidence in the learned model. Here, the integration of other machine learning techniques with easier understandable output, such as decision trees [SL91, WFH11], might be of interest. If these techniques do not produce adequate output, more experiments have to be performed to increase the confidence in the learned models, even though their contents are not transparent.

Furthermore, we do not specifically know if a new classifier is necessary for a particular software version or how *good* training data is selected. Further experiences are necessary to derive guidelines for future uses. Another open challenge is the selection of *negative* training data. Our evaluation has shown that the definition of negative test sets is challenging for test experts. Supporting the experts with newly found insights in future evaluations is of vital importance to further improve the

test case prioritization quality. Guidelines and experiences also reduce the initial effort to adapt our test case prioritization approach in different projects or domains due to an understanding of proper training data selection.

8.2 Future Work

While this thesis contributes approaches for black testing for software variants and versions with good results, we envision future work to improve our results. First and foremost, the presented techniques have to be applied to further real-life and large-scale case studies, especially the software variant related techniques. Further case studies will allow to generalize our findings.

Future Work for Testing of Software Variants. We envision that the presented SPL framework can be used as foundation for multi-objective optimization, similar to the our black-box test case selection approach. Test cases might be prioritized for individual product variants based on different objectives (e.g, weight metrics) based on search-based techniques, such as genetic algorithms. This avoids the manually definition of weighting factors. Instead, the optimization technique identifies optimal weightings itself by computing Pareto-optimal sets.

For our risk-based SPL testing technique, we envision more sophisticated approaches to define risk values for features, which influence our risk-based test case prioritization technique. Currently, we are only aware of one RBT approach for SPLs by Hartmann et al. [HvdLB14], which does not further state how feature-related risk values are computed or generated. Here, more complex approaches to automatically compute feature impact values will help to further reduce manual effort.

While both of our contributions for testing of software variants are based on test models, test models are usually not available for real-life industrial systems. Hence, we argue that a transformation of our test case prioritization approaches for SPLs from model-based to code-based techniques is an interesting research topic. This supports the applicability to currently existing systems in industry. To support the delta-oriented nature of our testing approaches, we argue that delta-oriented programming languages [SBB+10], such as DeltaJ [KHS+14], are an adequate starting point to realize incremental white-box SPL testing approaches for individual product variants. Instead of using test model vertices and edges, we envision a technique that uses control flow vertices and edges and prioritizes white-box test cases, e.g., provided as JUnit[1] scripts. JUnit test case prioritization has been performed for single-software systems [DRK04], which shows the demand for this type of approach, which as of now has not been realized for individual product variants in an SPL.

[1] JUnit framework for Java, current version: JUnit4, website: `http://junit.org`, date: March 29th 2017

Both of our test case prioritization approaches for software variants are influences by the ordering of product variants under test as we compute values based on previously tested variants. Thus, we propose that investigations are performed how the orderings of product variants influence our particular test case prioritization approaches and if certain orderings are to be preferred. Lity et al. [LAHTS17] investigated the orderings of product variants for regression testing of SPLs in general and showed that different orderings actually influence the results. Future work has to show if their findings can be mapped to our test case prioritization techniques for software variants.

Future Work for Testing of Software Versions. Our black-box testing approaches for software versions have shown that they outperform existing techniques, such as random, retest-all or manual test case prioritization. However, we also see a lot of potential for future work in terms of extensions.

Our multi-objective test case selection approach for software versions can be extended with additional objectives. Here, one interesting application in the future is a combination of both, black-box and white-box optimization objectives. For example, code coverage objectives could be integrated. Another adaption of our test case selection technique is to apply the multi-objective optimization to perform a test case prioritization instead of a test case selection. This requires different mutation operators to avoid the reduction of test sets. Instead, mutation and crossover need to create permutations of test sets, leading to different orderings. Similarly, objective functions need to be revised to compute the fitness of ordered test sets instead of selected subsets.

In addition, we envision a self-adapting machine learning approach to solve black-box testing of software versions. For example, the implementation of a *reinforcement learning* technique [KLM96], which adapts itself according to feedback, is one concept which we think can improve the currently applied test case prioritization. As of now, we have to relearn classifiers after a certain amount of iterations. This requires manual effort, i.e., selecting and adapting existing training data to reflect project changes. Instead of identifying the need for new classifiers and selecting training data manually, reinforcement learning supports the idea of self-training agents, e.g., based on trial-and-error. In case of software testing, machine learning algorithms can be supported with new failure findings. These could help to adjust the learned classification model to achieve even better failure finding rates. For example, if a test case that has received a high priority, but does not reveal new failures in several executions, it indicates that the ranked model needs adjustments to rank the test case lower. Currently, no reinforcement learning-based approaches for black-box testing have been proposed.

One challenge with introducing reinforcement learning for black-box testing is how to simulate different optimizations and provide enough data to learn from. Typically,

reinforcement learning thrives when it is repeated in a trial-and error fashion for different adjustments, e.g., inputs in a flight simulator to stabilize the airplane. This type of simulation is not possible in a testing scenario, where the execution of test cases might take several minutes and not all test cases are executed. To solve these issues, new concepts have to be developed and other data might be necessary to successfully realize a reinforcement learning approach for black-box systems.

Bibliography

[AA13] Z. Anwar and A. Ahsan. Multi-objective Regression Test Suite Optimization with Fuzzy Logic. In *Proceedings of the International Multi Topic Conference (INMIC)*, pages 95–100, 2013. 159, 161

[ABPS05] P. Avesani, C. Bazzanella, A. Perini, and A. Susi. Facing Scalability Issues in Requirements Prioritization with Machine Learning Techniques. In *Proceedings of the International Conference on Requirements Engineering (RE)*, pages 297–305, 2005. 165

[ACH+13] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based Model Slicing: A Survey. *ACM Computing Surveys*, 45(4):53:1–53:36, 2013. 69

[AFVB15] Hans-Martin Adorf, Michael Felderer, Martin Varendorff, and Ruth Breu. A Bayesian Prediction Model for Risk-Based Test Selection. In *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA*, pages 374–381, 2015. 117, 118

[AHLL+17] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proceedings of the International Workshop on Variability and Complexity in Software Design (VACE)*, 2017. 4, 90, 91

[AHTM+14] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based Prioritization in Software Product-line Testing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 197–206. ACM, 2014. 4, 79, 90, 91

[Aml00] Ståle Amland. Risk-based Testing: Risk Analysis Fundamentals and Metrics for Software Testing Including a Financial Application Case Study. *Journal of Systems and Software*, 53:287–295, 2000. 97, 98, 99, 109, 134, 140, 141, 160

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 12, 17, 137, 185, 197

[ASW+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions Using Feature-aware Verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375, 2011. 42

[ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A Systematic Review of Search-Based Testing for Non-Functional System Properties. *Information and Software Technology*, 51(6):957 – 976, 2009. 158

[ATLK12] D. Agarwal, D.E. Tamir, M. Last, and A. Kandel. A Comparative Study of Artificial Neural Networks and Info-Fuzzy Networks as Automated Oracles in Software Testing. *Proceedings of the International Conference on Systems, Man and Cybernetics (SMC)*, 42(5):1183–1193, 2012. 190, 192

[AWSE16] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Test Case Prioritization of Configurable Cyber-Physical Systems with Weight-Based Search Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1053–1060. ACM, 2016. 4, 91, 93

[Bas07] Len Bass. *Software Architecture in Practice*. Pearson Education India, 2007. 14

[BERL13] G. Buchgeher, C. Ernstbrunner, R. Ramler, and M. Lusser. Towards Tool-Support for Test Case Selection in Manual Regression Testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 74–79, 2013. 163

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 63

[BGK04] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004. 127

[Bin99] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1999. 161

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 166

[BK14] Harsh Bhasin and Esha Khanna. Neural Network Based Black Box Testing. *Software Engineering Notes*, 39(2):1–6, 2014. 190, 193

[BKY12] Xiaoying Bai, Ron S Kenett, and Wei Yu. Risk Assessment and Adaptive Group Testing of Semantic Web Services. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 22(05):595–620, 2012. 118, 119

[BL14] Hauke Baller and Malte Lochau. Towards Incremental Test Suite Optimization for Software Product Lines. In *FOSD*, pages 30–36. ACM, 2014. 91, 93

[BLB08] L.C. Briand, Y. Labiche, and Z. Bawar. Using Machine Learning to Refine Black-Box Test Specifications and Test Suites. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 135–144, 2008. 190, 193

[BLC13] Lionel C. Briand, Yvan Labiche, and Kathy Chen. A Multi-objective Genetic Algorithm to Rank State-Based Test Cases. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 66–80, 2013. 4, 159, 161

[BLLS14] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-objective Test Suite Optimization for Incremental Product Family Testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 303–312, 2014. 91, 93

[BPM04] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. *SIGKDD Explor. Newsl.*, 6(1):20–29, 2004. 167

[Bro06] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 33–42, 2006. 13

[BSS02] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness Function Design To Improve Evolutionary Structural Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1329–1336, 2002. 158

[BT96] Tobias Blickle and Lothar Thiele. A Comparison of Selection Schemes Used in Evolutionary Algorithms. *Evolutionary Computation*, 4(4):361–394, 1996. 128, 129

[BVSF04] Ricardo Barandela, Rosa Maria Valdovinos, José Salvador Sánchez, and Francesc J. Ferri. The Imbalanced Training Sample Problem: Under or over Sampling? In *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops, SSPR and SPR August 18-20, 2004*, page 806, 2004. 167

[Cat11] Cagatay Catal. Software Fault Prediction: A Literature Review and Current Trends. *Expert Systems with Applications*, 38(4):4626–4636, 2011. 192

[CD09] Cagatay Catal and Banu Diri. A Systematic Review of Software Fault Prediction Studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009. 192

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000. 26, 27, 28

[Cen77] Yair Censor. Pareto Optimality in Multiobjective Problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977. 128

[CG09] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Pearson Education, 2009. 18

[CH08] Pavan Kumar Chittimalli and Mary Jean Harrold. Regression Test Selection on System Requirements. In *Proceedings of the India Software Engineering Conference (ISEC)*, pages 87–96, 2008. 159, 160

[CHS10] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22, 2010. 28, 32, 102, 103, 197, 199

[CKMRM02] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. Technical report, Department of Computing Science, University of Glasgow, United Kingdom, May 2002. 42

[CLWK00] Xia Cai, M. R. Lyu, Kam-Fai Wong, and Roy Ko. Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 372–379, 2000. 13, 14

[Coe00] Carlos A. Coello. An Updated Survey of GA-based Multiobjective Optimization Techniques. *ACM Computing Surveys*, 32(2):109–143, 2000. 128, 142

[CPS02] Yanping Chen, Robert L Probert, and D Paul Sims. Specification-Based Regression Test Selection with Risk Analysis. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2002. 4, 19, 117, 118, 159, 160

[CRK10] Alexander P. Conrad, Robert S. Roos, and Gregory M. Kapfhammer. Empirically Studying the Role of Selection Operators During Search-Based Test Suite Prioritization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1373–1380, 2010. 129

[CST00] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000. 168, 169

[CW14] Gary Chen and Pei-Qi Wang. Test Case Prioritization in a Specification-based Testing Environment. *Journal of Software*, 9(8), 2014. 189, 190

[Dav87] L. Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufman Publishers, Inc.,Los Altos, CA, 1987. 126, 157

[Deb01] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., 2001. 126, 127

[DH07] Pedro A. Diaz-Gomez and Dean F. Hougen. Initial Population for Genetic Algorithms: A Metric Approach. In *Proceedings of the 2007 International Conference on Genetic and Evolutionary Methods, GEM*, pages 43–49, 2007. 126

[Die95] Tom Dietterich. Overfitting and Undercomputing in Machine Learning. *ACM Computing Surveys*, 27(3):326–327, 1995. 167

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the Analysis of the (1+1) Evolutionary Algorithm. *Theoretical Computer Science*, 276(1-2):51–81, 2002. 160, 161

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978. 112, 116

[DMSNdCMC+10] P.A. Da Mota Silveira Neto, I. do Carmo Machado, Y.C. Cavalcanti, E.S. de Almeida, V.C. Garcia, and S.R. de Lemos Meira. A Regression Testing Approach for Software Product Lines Architectures. In *SBCARS*, pages 41–50, 2010. 91, 94

[DN11] Juan J. Durillo and Antonio J. Nebro. jMetal: A Java Framework for Multi-objective Optimization. *Advances in Engineering Software*, 42(10):760–771, 2011. 145

[DPAM02] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. 128, 143, 145, 146, 147, 162

[DPC+13] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 10:1–10:7. ACM, 2013. 4, 90, 91, 92

[DRK04] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 113–124, 2004. 201

[dSdMPdB11] L.S. de Souza, P.B.C. de Miranda, R.B.C. Prudencio, and F.A. de Barros. A Multi-objective Particle Swarm Optimization for Test Case Selection Based on Functional Requirements Coverage and Execution Effort. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 245–252, 2011. 159, 161

[DSLL13] Michael Dukaczewski, Ina Schaefer, Remo Lachmann, and Malte Lochau. Requirements-based Delta-oriented SPL Testing. In *Proceedings of the International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, pages 49–52, 2013. 28

[dSPdAB14] L. S. de Souza, R. B. C. Prudêncio, and F. d. A. Barros. A Comparison Study of Binary Multi-Objective Particle Swarm Optimization Approaches for Test Case Selection. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, pages 2164–2171, 2014. 159, 161

[EBA+11] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proceedings of the International Conference on Information Technology: New Generations*, pages 291–298, 2011. 91, 92

[EE15] Edward Dunn Ekelund and Emelie Engström. Efficient Regression Testing Based on Test History: An Industrial Evaluation. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, page 9. IEEE Computer Society, 2015. 159, 160

[ELR+14] Gencer Erdogan, Yan Li, Ragnhild Kobro Runde, Fredrik Seehusen, and Ketil StØlen. Approaches for the Combined Use of Risk Analysis and Testing: A Systematic Literature Review. *International Journal on Software Tools and Technology Transfer (STTT)*, pages 627–642, 2014. 97, 99, 117, 120, 134, 141, 197

[Ema05] Khaled El Emam. *The ROI from Software Quality*. Auerbach Publications - Taylor & Francis Group, 2005. 41

[EMR01] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2001. 20, 21, 92, 162, 178, 191

[Eng10] Emelie Engström. *Exploring Regression Testing and Software Product Line Testing - Research and State of Practice*. Lic dissertation, Lund University, May 2010. 3, 5, 22, 53

[EP15] E. Engström and K. Petersen. Mapping Software Testing Practice with Software Testing Research - SERP-test Taxonomy. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, 2015. 166

[ER11] Emelie Engström and Per Runeson. Software Product Line Testing – A Systematic Mapping Study. *Information and Software Technology*, 53:2–13, 2011. 25, 90

[ERL11] E. Engström, P. Runeson, and A. Ljung. Improving Regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 367–376. IEEE, 2011. 4, 21, 74, 82, 189, 190

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*, 52:14–30, 2010. 14, 19, 20, 98, 125, 133, 138, 149, 189, 198

[EYHB15] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. Empirical Evaluation of Pareto Efficient Multi-objective Regression Test Case Prioritisation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 234–245, 2015. 4, 159, 162

[Faw06] Tom Fawcett. An introduction to {ROC} analysis. *Pattern Recognition Letters*, 27(8):861 – 874, 2006. {ROC} Analysis in Pattern Recognition. 147

[FGG97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 29(2):131–163, 1997. 169

[FHBM12] Michael Felderer, Christian Haisjackl, Ruth Breu, and Johannes Motz. Integrating Manual and Automatic Risk Assessment for Risk-Based Testing. *Software Quality. Process Automation in Software Development*, pages 159–180, 2012. 118, 119, 133

[FHPB14] Michael Felderer, Christian Haisjackl, Viktor Pekar, and Ruth Breu. A Risk Assessment Framework for Software Testing. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 292–308. 2014. 135

[FIMN07] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I Malik, and Aamer Nadeem. An Approach for Selective State Machine Based Regression Testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 44–52. ACM, 2007. 158

[FKAP09] Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi, and S. Parsa. Prioritizing Test Cases for Resource Constraint Environments using Historical Test Case Performance Data. In *Proceedings of the International Conference on Computer Science and Information Technology (ICCSIT)*, pages 190–195. IEEE, 2009. 4, 21, 74, 189, 190

[FR14] Michael Felderer and Rudolf Ramler. Integrating Risk-Based Testing in Industrial Test Processes. *Software Quality Journal*, 22(3):543–575, 2014. 40

[FS97] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. 195

[FS14] Michael Felderer and Ina Schieferdecker. A Taxonomy of Risk-based Testing. *International Journal on Software Tools and Technology Transfer (STTT)*, 16(5):559–568, 2014. 97, 98, 99, 117, 120, 140

[Fus09] Tadayoshi Fushiki. Estimation of Prediction Error by using K-fold Cross-Validation. *Statistics and Computing*, 21(2):137–146, 2009. 178

[FvBK+91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection based on Finite State Models. *IEEE Transactions on Software Engineering (TSE)*, 17(6):591–603, 1991. 158

[GG05] Cyril Goutte and Eric Gaussier. *A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation*, pages 345–359. Springer Berlin Heidelberg, 2005. 148

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001. 19

[GKN12] Kehan Gao, Taghi M. Khoshgoftaar, and Amri Napolitano. A Hybrid Approach to Coping with High Dimensionality and Class Imbalance for Software Defect Prediction. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, pages 281–288, 2012. 167, 168

[GM00] David Greenhalgh and Stephen Marshall. Convergence Criteria for Genetic Algorithms. *SIAM Journal on Computing*, 30(1):269–282, 2000. 132

[Gon08] Iker Gondra. Applying Machine Learning to Software Fault-Proneness Prediction. *Journal of Systems and Software*, 81(2):186–195, 2008. 190, 192

[HAB11] H. Hemmati, A. Arcuri, and L. Briand. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 327–336, 2011. 158

[HAB13] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving Scalable Model-based Testing Through Test Case Diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):6:1–6:42, 2013. 159, 160, 161

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, Juni 1987. 16, 44

[Har11] M. Harman. Making the Case for MORTO: Multi Objective Regression Test Optimization. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 111–114, March 2011. 161

[HDC14] C. Hettiarachchi, Hyunsook Do, and Byoungju Choi. Effective Regression Testing Using Requirements and Risks. In *Proceedings of the International Conference on Software Security and Reliability (SERE)*, pages 157–166, 2014. 118, 119

[HDC16] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi. Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, 69:1–15, 2016. 118, 119

[HFM15] Hadi Hemmati, Zhihan Fang, and Mika V. Mäntylä. Prioritizing Manual Test Cases in Traditional and Rapid Release Environments. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 1–10, 2015. 21, 190, 191

[HGCM15] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The Art of Testing Less without Sacrificing Quality. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015. to appear. 159, 160

[HGLJ13] M. Huang, S. Guo, X. Liang, and X. Jiao. Research on Regression Test Case Selection based on Improved Genetic Algorithm. In *Proceedings of the International Conference on Computer Science and Network Technology (ICCSNT)*, pages 256–259, 2013. 159, 162

[HJZ15] M. Harman, Y. Jia, and Y. Zhang. Achievements, Open Problems and Challenges for Search Based Software Testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 1–12, 2015. 121, 126, 158

[HLS99] Mary Jean Harrold, Donglin Liang, and Saurabh Sinha. An Approach To Analyzing and Testing Component-Based Systems. In *Workshop on Testing distributed Component-Based Systems*, 1999. 13

[HM10] M. Harman and P. McMinn. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering (TSE)*, 36(2):226–247, 2010. 158

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. 126

[HPH12] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. A History-Based Cost-Cognizant Test Case Prioritization Technique in Regression Testing. *Journal of Systems and Software*, 85(3):626 – 637, 2012. 190, 191

[HPH+16] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 523–534, 2016. 163, 189

[HS11] Basima Hani Hasan and Moutaz Saleh Mustafa Saleh. Evaluating the Effectiveness of Mutation Operators on the Behavior of Genetic Algorithms Applied to Non-deterministic Polynomial Problems. *Informatica (Slovenia)*, 35(4):513–518, 2011. 130

[HT03] David Harel and P. S. Thiagarajan. Message Sequence Charts. In *In UML for Real: Design of Embedded Real-Time Systems*, pages 77–105, 2003. 37

[HvdLB14] Herman Hartmann, Frank van der Linden, and Jan Bosch. Risk Based Testing for Software Product Line Engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 227–231, 2014. 97, 100, 115, 117, 118, 120, 197, 201

[HvKS08] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. Code Reuse in Open Source Software. *Management Science*, 54(1):180–193, 2008. 13

[IR05] J. Itkonen and K. Rautiainen. Exploratory Testing: A Multiple Case Study. In *International Symposium on Empirical Software Engineering (ISESE)*, pages pp. 84–93, Nov 2005. 188

[ISO09] ISO/DIS 26262-1 - Road vehicles - Functional safety, 2009. 199

[JCM07] Y. Jiang, B. Cukic, and T. Menzies. Fault Prediction using Early Lifecycle Data. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 237–246, 2007. 190, 192

[JH08] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*, pages 249–258, 2008. 116, 158

[JH11] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2011. 112

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55, 2012. 4, 42, 90

[JM00] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. 171

[Joa02] Thorsten Joachims. Optimizing Search Engines Using Clickthrough Data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 133–142. ACM, 2002. 169, 177, 180, 194, 198, 200, 247

[Kan91] A. Kandel. *Fuzzy Expert Systems*. Taylor & Francis, 1991. 119

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute - Carnegie Mellon University, November 1990. 26, 29

[KCS06] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-Objective Optimization using Genetic Algorithms: A Tutorial. *Reliability Engineering & System Safety*, 91(9):992 – 1007, 2006. 126, 127, 129, 142, 143, 157

[KE95] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Neural Networks. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948, 1995. 161

[KGN12] Taghi M. Khoshgoftaar, Kehan Gao, and Amri Napolitano. An Empirical Study of Feature Ranking Techniques for Software Quality Prediction. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 22(2):161–183, 2012. 168

[KHE11] Johannes Kloos, Tanvir Hussain, and Robert Eschbach. Risk-Based Testing of Safety-Critical Embedded Systems driven by Fault Tree Analysis. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 26–33, 2011. 118, 119

[KHS+14] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java platform (PPPJ)*, pages 63–74, 2014. 28, 201

[Kim10] Jin-Lee Kim. Examining the Relationship Between Algorithm Stopping Criteria and Performance Using Elitist Genetic Algorithm. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 3220–3227, 2010. 132

[Kin09] Davis E. King. Dlib-ml: A Machine Learning Toolkit. *Journal on Machine Learning Research*, 10:1755–1758, 2009. 177

[KKT07] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based Test Prioritization Heuristic Methods and Their Evaluation. In *Proceedings of the International Workshop on Advances in Model-based Testing (A-MOST)*, pages 34–43. ACM, 2007. 21

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 202

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin.

Aspect-Oriented Programming, pages 220–242. Springer Berlin Heidelberg, 1997. 28

[Kot07] Sotiris B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007. 166, 168

[KP02] Jung-Min Kim and A. Porter. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 119–129, 2002. 21

[KRS15] Alexander Knapp, Markus Roggenbach, and Bernd-Holger Schlingloff. Automating Test Case Selection in Model-Based Software Product Line Development. *International Journal of Software and Informatics(IJSI)*, 9(2):153–175, 2015. 91, 93

[KTS10] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, pages 571–579, 2010. 134

[Lac15] Hartmut Lackner. *Model-Based Product Line Testing: Sampling Configurations for Optimal Fault Detection*, pages 238–251. Springer International Publishing, 2015. 91, 93

[LAHTS17] Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. Optimizing product orders using graph algorithms for improving incremental product-line analysis. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 60–67, 2017. 202

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015. 200

[LBL+17] Remo Lachmann, Simon Beddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. Risk-based Integration Testing of Software Product Lines. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 52–59, 2017. 7, 97, 100, 106, 112, 114, 116, 197

[Leh80] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 3, 13, 17

[LFN+17] Remo Lachmann, Michael Felderer, Manuel Nieke, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Multi-Objective Black-Box Test Case Selection for System Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2017. 7, 125, 133, 134, 136, 198

[LHH07] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering (TSE)*, 33(4):225–237, 2007. 21

[LHJFC+14] Roberto Erick Lopez-Herrejon, Javier Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. A Parallel Evolutionary Algorithm for Prioritized Pairwise Testing of Software Product Lines. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1255–1262, 2014. 91, 92

[LKM+99] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Iñaki Inza, and S. Dizdarevic. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13(2):129–170, 1999. 126

[LLAH+16] Remo Lachmann, Sascha Lity, Mustafa Al-Hajjaji, Franz E. Fürchtegott, and Ina Schaefer. Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, 2016. 6, 48, 53, 56, 66, 68, 69, 75, 77, 197

[LLH03] Wen-Yang Lin, Wen-Yung Lee, and Tzung-Pei Hong. Adapting Crossover and Mutation Rates in Genetic Algorithms. *Journal of Information Science and Engineering (JISE)*, 19(5):889–903, 2003. 131

[LLL+14] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-Oriented Model-Based Integration Testing of Large-Scale Systems. *Journal of Systems and Software*, 91:63–84, 2014. 4, 28, 35, 45, 64, 89, 91, 93, 116, 199

[LLL+15] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. Delta-Oriented Test Case Prioritization for Integration Testing of Software Product Lines. In

Proceedings of the International Software Product Line Conference (SPLC), pages 81–90, 2015. 6, 36, 53, 56, 64, 74, 75, 76, 89, 116, 197

[LLLS12] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study. Technical report, TU Braunschweig, 2012. 7, 29, 42, 43, 44, 45, 46, 47, 48, 49, 64, 89, 112, 116, 145, 197

[LLS+17] Remo Lachmann, Sascha Lity, Sandro Schulze, Christoph Seidl, and Ina Schaefer. A Framework for Model-Based Test Case Prioritization for Software Product Lines. *Software & Systems Modeling (SOSYM)*, 2017. (submitted.). 6, 53, 54, 62, 73, 76, 85, 197

[LLSG12] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. Delta-oriented Model-based SPL Regression Testing. In *Proceedings of the International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, 2012. 42, 44

[LLTY15] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. Many-Objective Evolutionary Algorithms: A Survey. *ACM Computing Surveys*, 48(1):13:1–13:35, 2015. 157, 164

[LMTS16] Sascha Lity, Thomas Morbach, Thomas Thüm, and Ina Schaefer. Applying Incremental Model Slicing to Product-Line Regression Testing. In *ICSR*, 2016. (to appear). 91, 93

[LOGS11] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. *Software Quality Journal*, pages 1–38, 2011. 4, 90

[LPBM12] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing Test Cases with String Distances. *Automated Software Engineering*, 19(1):65–95, 2012. 190, 191

[LS14] Remo Lachmann and Ina Schaefer. Towards Efficient and Effective Testing in Automotive Software Development. In *Proceedings of the Automotive Software Engineering Workshop*, 2014. 3

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-based Testing of Delta-oriented Software Product Lines. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 67–82, 2012. 28, 35, 44, 57

[LSN+16] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. System-Level Test Case Prioritization Using Machine Learning. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, 2016. 7, 89, 112, 165, 170

[LV04] William E. Lewis and Gunasekaran Veerapillai. *Software Testing and Continuous Quality Improvement, Second Edition*. Auerbach Publications, 2004. 14, 163, 185

[LW89] Hareton K. N. Leung and Lee White. Insights into Regression Testing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 60–69, 1989. 17, 18, 20

[LW00] Eckard Lehmann and Joachim Wegener. Test Case Design by Means of the CTE-XL. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, 2000. 160

[McM11] Phil McMinn. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163. IEEE Computer Society, 2011. 95, 121, 126, 158

[MG95] Brad L Miller and David E Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex systems*, 9(3):193–212, 1995. 129

[MHD15] D. Mondal, H. Hemmati, and S. Durocher. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 1–10, 2015. 4, 159, 162

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996. 126, 129, 130, 146

[MLD+09] Tobias Müller, Malte Lochau, Stefan Detering, Falko Saust, Henning Garbers, Lukas Märtin, Thomas Form, and Ursula Goltz. A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance. Technical Report Technical Report 2009-06, Technische Universität Braunschweig, 2009. 41

[MM13] D. N. Mudaliar and N. K. Modi. Unraveling Travelling Salesman Problem by Genetic Algorithm using m-crossover operator. In *Signal Processing Image Processing Pattern Recognition (ICSIPR), 2013 International Conference on*, pages 127–130, 2013. 129

[MMCDA14] Ivan Do Carmo Machado, John D. Mcgregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology*, 56(10):1183–1199, 2014. 3, 53, 90

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering (TSE)*, 26(1):70–93, Jan 2000. 15

[Muc07] Henry Muccini. Using model Differencing for Architecture-level Regression Testing. In *Proceedings of the Conference on Software Engineering and Advanced Applications (SEAA)*, 2007. 91, 92

[MvdH03] H Muccini and A. van der Hoek. Towards Testing Product Line Architectures. *Electronic Notes in Theoretical Computer Science*, 82(6):99 – 109, 2003. TACoS'03. 90

[MW47] Henry B Mann and Donald R Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The annals of mathematical statistics*, pages 50–60, 1947. 87, 156, 163, 182

[MYC12] M. Song M. Yoon, E. Lee and B. Choi. A Test Case Prioritization through Correlation of Requirement and Risk. *Journal of Software Engineering and Applications (JSEA)*, pages 823–835, 2012. 118, 120

[NH15] T. B. Noor and H. Hemmati. A Similarity-Based Approach for Test Case Prioritization using Historical Failure Data. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68, 2015. 79

[NWF+15] Tadahiro Noguchi, Hironori Washizaki, Yoshiaki Fukazawa, Atsutoshi Sato, and Kenichiro Ota. History-Based Test Case Prioritization for Black Box Testing Using Ant Colony Optimization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 1–2, 2015. 190, 191

[NZR09] L. Naslavsky, H. Ziv, and D. J. Richardson. A Model-Based Regression Test Selection Technique. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 515–518, Sept 2009. 158

[OB88] T. J. Ostrand and M. J. Balcer. The Category-partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM*, 31(6):676–686, 1988. 193

[OHB99] Gabriela Ochoa, Inman Harvey, and Hilary Buxton. On Recombination and Optimal Mutation Rates. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 488–495. Morgan Kaufmann Publishers Inc., 1999. 131

[OMG15] OMG. UML, Version 2.5., OMG Unified Modeling Language, `http://www.omg.org/spec/UML/2.5/`, 2015. 16, 160

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*. Berlin, Heidelberg: Springer-Verlag, 2010. 42

[OZLG11] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 6:1–6:8, 2011. 42, 90

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005. 3, 5, 22, 23, 25, 26

[PC95] P.W. Poon and J.N. Carter. Genetic Algorithm Crossover Operators for Ordering Applications. *Computers & Operations Research*, 22(1):135 – 147, 1995. 129

[PKK+15] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf. SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 852–857, 2015. 33, 62

[Pre97] Christian Prehofer. *Feature-Oriented Programming: A Fresh Look at Objects*, pages 419–443. Springer Berlin Heidelberg, 1997. 28

[PS06] Alan Piszcz and Terence Soule. Genetic Programming: Optimal Population Sizes for Varying Complexity Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 953–954, 2006. 126

[PSA13] Anna Perini, Angelo Susi, and Paolo Avesani. A Machine Learning Approach to Software Requirements Prioritization. *IEEE Transactions on Software Engineering (TSE)*, 39(4):445–461, 2013. 190, 193

[PSS+16] José A. Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. Multi-Objective Test Case Prioritization in Highly Configurable Systems: A Case Study. *Journal of Systems and Software*, 122:287 – 310, 2016. 4, 91, 92

[QCR08] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *ISSTA*, pages 75–86, 2008. 91, 92

[QNXZ07] Bo Qu, Changhai Nie, Baowen Xu, and Xiaofang Zhang. Test Case Prioritization for Black Box Testing. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pages 465–474, 2007. 4, 190, 191

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. 193

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–118, 1992. 12

[RBT13] Erik Rogstad, Lionel Briand, and Richard Torkar. Test Case Selection for Black-box Regression Testing of Database Applications. *Information and Software Technology*, 55(10):1781–1795, 2013. 4, 159, 160

[RFP13] O. Roeva, S. Fidanova, and M. Paprzycki. Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling. In *Computer Science and Information Systems (FedCSIS), Federated Conference on*, pages 371–376, 2013. 126, 147

[RG99] B. Rosner and D. Grove. Use of the Mann–Whitney U-Test for Clustered Data. *Statistics in Medicine*, 18(11):1387–1400, 1999. 156

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–184. ACM, 1994. 4, 6, 125, 126, 133, 149

[RH96] G. Rothermel and M.J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering (TSE)*, 22(8):529–551, 1996. 19

[RH97] Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, April 1997. 19, 133, 165

[RH09] Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. 89, 157, 187

[RMP07] Sacha Reis, Andreas Metzger, and Klaus Pohl. Integration Testing in Software product Line Engineering: A Model-Based Technique. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 321–335. Springer-Verlag Berlin Heidelberg, 2007. 91, 94

[RUCH99] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 179–188, 1999. 90

[RUCH01] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering (TSE)*, Vol.27 No.10:929–948, 2001. 20, 21, 74, 112, 113, 178

[SA13] A. S. Sayyad and H. Ammar. Pareto-Optimal Search-Based Software Engineering (POSBSE): A Literature Survey. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), International Workshop on*, pages 21–27, 2013. 127, 157

[Saa08] Thomas L Saaty. Decision making with the analytic hierarchy process. *International journal of services sciences*, 1(1):83–98, 2008. 120

[Sar07] Siamak Sarmady. An Investigation on Genetic Algorithm Parameters. *School of Computer Science, Universiti Sains Malaysia*, 2007. 126

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. 194

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91, 2010. 28, 201

[SBWO14] Hema Srikanth, Sean Banerjee, Laurie Williams, and Jason Osborne. Towards the Prioritization of System Test Cases. *Journal of Software Testing, Verification and Reliability*, 24(4):320–337, 2014. 190, 191

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft press, 2004. 9

[SCPB04] Martín Safe, Jessica Carballido, Ignacio Ponzoni, and Nélida Brignole. *On Stopping Criteria for Genetic Algorithms*, pages 405–413. Springer Berlin Heidelberg, 2004. 132, 142

[Sel07] B. Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Proceedings of the IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–9, 2007. 62

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016. 194

[SHT06] P. Y. Schobbens, P. Heymans, and J. C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the International Conference on Requirements Engineering (RE)*, pages 139–148, 2006. 27

[SL91] S. Rasoul Safavian and David A. Landgrebe. A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man, and Cybernetics (TSMC)*, 21(3):660–674, 1991. 188, 200

[SLS11] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam.* Rocky Nook, 3rd edition, 2011. 10, 11, 12, 50

[SMP08] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl. An Automated Technique for Risk-based Test Case Generation and Prioritization. In *AST 2008*, pages 67–70, 2008. 118, 120

[Sne07] H. M. Sneed. Testing against Natural Language Requirements. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 380–387, 2007. 12, 18, 81, 144, 165, 174

[Som10] Ian Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, USA, 9th edition, 2010. 3, 9, 11, 12, 39, 45, 47, 90

[SP94] M. Srinivas and L. M. Patnaik. Genetic Algorithms: A Survey. *Computer*, 27(6):17–26, June 1994. 126, 142

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools and Technology Transfer (STTT)*, 14(5):477–495, 2012. 28

[Tip01] Michael E. Tipping. Sparse Bayesian Learning and the Relevance Vector Machine. *Journal on Machine Learning Research*, 1:211–244, 2001. 169

[TKK96] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. Multi-Objective Optimization by Genetic Algorithms: A Review. In *Proceedings of International Conference on Evolutionary Computation (ICEC)*, pages 517–522, 1996. 146

[UKB10] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental Test Generation for Software Product Lines. *IEEE Transactions on Software Engineering (TSE)*, 36(3):309–322, 2010. 4, 91, 94

[UL07] Mark Utting and Bruno Legeard. *Practical Model-based Testing*. Morgan Kaufmann, 2007. 5, 14, 16, 30, 31, 34, 44, 45, 56, 89, 197, 198

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook.org, 2013. 63

[VBM15] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. Delta-Oriented FSM-Based Testing. In *ICFEM*, volume 9407 of *LNCS*, pages 366–381. Springer, 2015. 4, 91, 94

[WAG13] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Minimizing Test Suites in Software Product Lines Using Weight-based Genetic Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1493–1500, 2013. 91, 94

[WAG15] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Cost-Effective Test Suite Minimization in Product Lines Using Search Techniques. *Journal of Systems and Software*, 103:370 – 391, 2015. 4, 91, 94

[WCO03] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. *UML-Based Integration Testing for Component-Based Software*, pages 251–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 13

[Wei04] Gary M. Weiss. Mining with Rarity: A Unifying Framework. *ACM SIGKDD Explorations Newsletter - Special issue on learning from imbalanced datasets*, 6(1):7–19, 2004. 167

[Wey98] Elaine J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, 1998. 4, 13, 39

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011. 166, 168, 178, 187, 194, 200

[WH06] Alison Watkins and Ellen M. Hufnagel. Evolutionary Test Data Generation: A Comparison of Fitness Functions. *Software: Practice and Experiences*, 36(1):95–116, 2006. 127

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage Metrics for Requirements-based Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 25–36. ACM, 2006. 12, 40, 137

[WRSS17] David Wille, Tobias Runge, Christoph Seidl, and Sandro Schulze. Extractive Software Product Line Engineering Using Model-Based Delta Module Generation. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 36–43, 2017. 33, 62, 199

[WSS16] David Wille, Sandro Schulze, and Ina Schaefer. Variability mining of state charts. In Christoph Seidl and Leopoldo Teixeira, editors, *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 63–73. ACM, 2016. 199

[WT11] Josh L. Wilkerson and Daniel R. Tauritz. A Guide for Fitness Function Design. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 123–124, 2011. 127

[XES+92] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gal, S. Katsikas, and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1992. 158

[XSWS09] L. Xiaosong, L. Shushi, C. Wenjun, and F. Songjiang. The Application of Risk Matrix to Software Project Risk Management. In *International Forum on Information Technology and Applications (IFITA)*, volume 2, pages 480–483, 2009. 135

[YH07a] Shin Yoo and Mark Harman. Pareto Efficient Multi-objective Test Case Selection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 140–150. ACM, 2007. 4, 126, 159, 162, 200

[YH07b] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2007. 4, 6, 18, 19, 90, 125, 144, 165, 189, 197

[YHTS09] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212. ACM, 2009. 21, 190, 193

[YL12] Yuen-Tak Yu and Man Fai Lau. Fault-based Test Suite Prioritization for Specification-based Testing. *Information and Software Technology*, 54(2):179–202, 2012. 190, 192

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. 137

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical report, Swiss Federal Institute of Technology, Computer Engineering and Networks Laboratory, 2001. 161

[ZWRS07] J. Zheng, L. Williams, B. Robinson, and K. Smiley. Regression Test Selection for Black-box Dynamic Link Library Components. In *Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques. IWICSS.*, pages 9–9, 2007. 159, 161

Part IV

Appendix

A Evaluation of SPL Framework

Table A.1: Pairwise p-Values for SPL Framework Instances

First TCP	Second TCP	p-Value
Random	CB	0.346034302466435
Random	ComB	0.473895598667925
Random	DB	0.000274944920429029
Random	CB and MD	0.0000756752009714705
Random	ComB and DB	0.00593151979026971
Random	SB and DB	0.0175670504999475
Random	SB and BCB	0.0175670504999475
Random	CB and DB	0.00593151979026971
Random	BCB, CB and DB	0.00417509526593009
CB	ComB	1
CB	DB	0.18350676188255
CB	CB and MD	0.0594813462804308
CB	ComB and DB	0.227672125849788
CB	SB and DB	0.0103747743736024
CB	SB and BCB	0.0128573374211115
CB	CB and DB	0.282674314837381
CB	BCB, CB and DB	0.23506255691367
ComB	DB	0.254227272328748
ComB	CB and MD	0.11340331293692
ComB	ComB and DB	0.274271216222805
ComB	SB and DB	0.0175670504999475
ComB	SB and BCB	0.0261585397960082
ComB	CB and DB	0.26612657639469
ComB	BCB, CB and DB	0.282674314837381

BCB = Behavior Component-Based **CB** = Component-based, **SB** = Signal-based, **DB** = Dissimilarity-based, **MB** = Meta-Data

Table A.2: Pairwise p-Values for SPL Framework Instances (cont.)

First TCP	Second TCP	p-Value
DB	CB and MD	0.925995689195583
DB	ComB and DB	0.838059800952748
DB	SB and DB	0.0000573101378336432
DB	SB and BCB	0.000070689379834867
DB	CB and DB	0.668930879611627
DB	BCB, CB and DB	0.809125348441329
CB and MD	ComB and DB	0.651046775454994
CB and MD	SB and DB	0.0000550496253588344
CB and MD	SB and BCB	00000645874376263366
CB and MD	CB and DB	0.597712283811784
CB and MD	BCB, CB and DB	0.734435452594774
ComB and DB	SB and DB	0.000603568142031714
ComB and DB	SB and BCB	0.000693050894381199
ComB and DB	CB and DB	0.880135745456665
ComB and DB	BCB, CB and DB	1
SB and DB	SB and BCB	0.939842517724922
SB and DB	CB and DB	0.000524936346778489
SB and DB	BCB, CB and DB	0.000524936346778489
SB and BCB	CB and DB	0.000603568142031714
SB and BCB	BCB, CB and DB	0.000524936346778489
CB and DB	BCB, CB and DB	0.865226720063674

BCB = Behavior Component-Based **CB** = Component-based, **SB** = Signal-based, **DB** = Dissimilarity-based, **MB** = Meta-Data

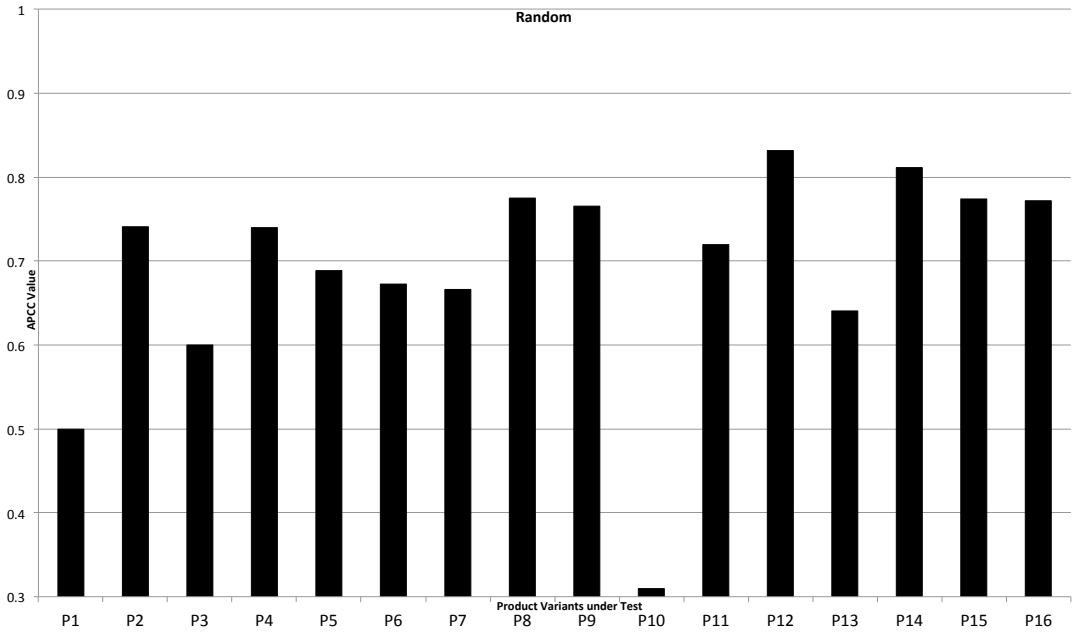

Figure A.1: APCC for BCS with Random Test Case Prioritization

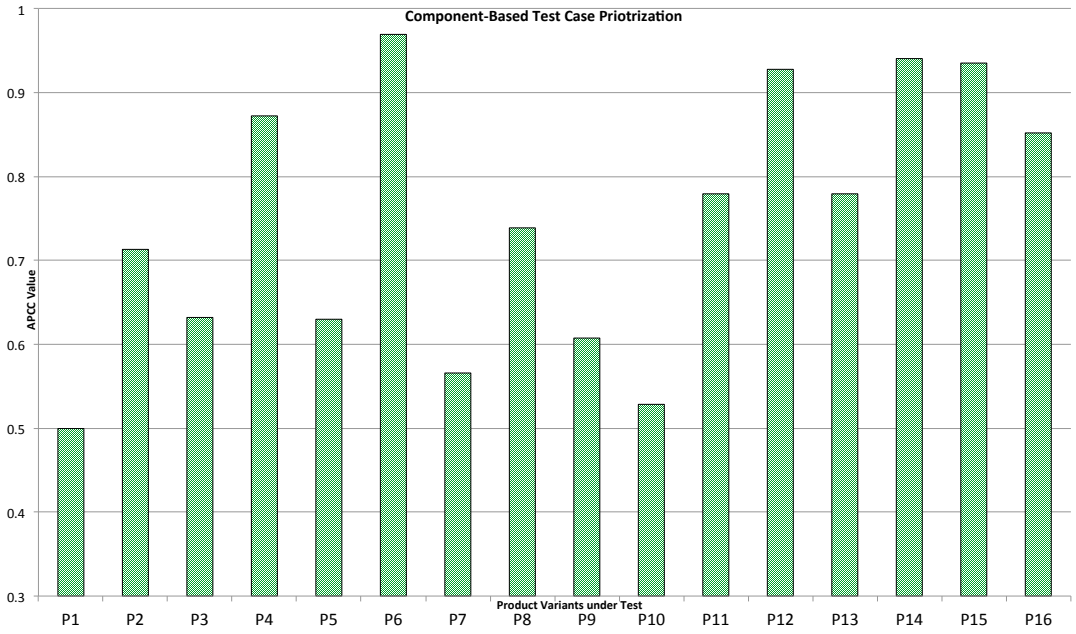

Figure A.2: APCC for BCS with Component-Based Test Case Prioritization

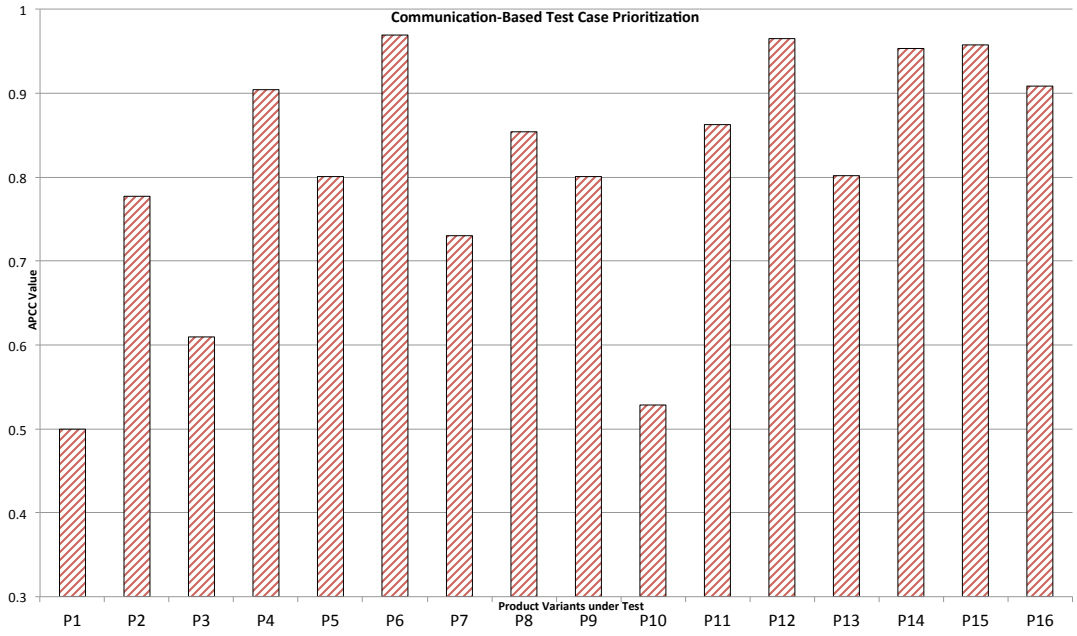

Figure A.3: APCC for BCS with Communication-Based Test Case Prioritization

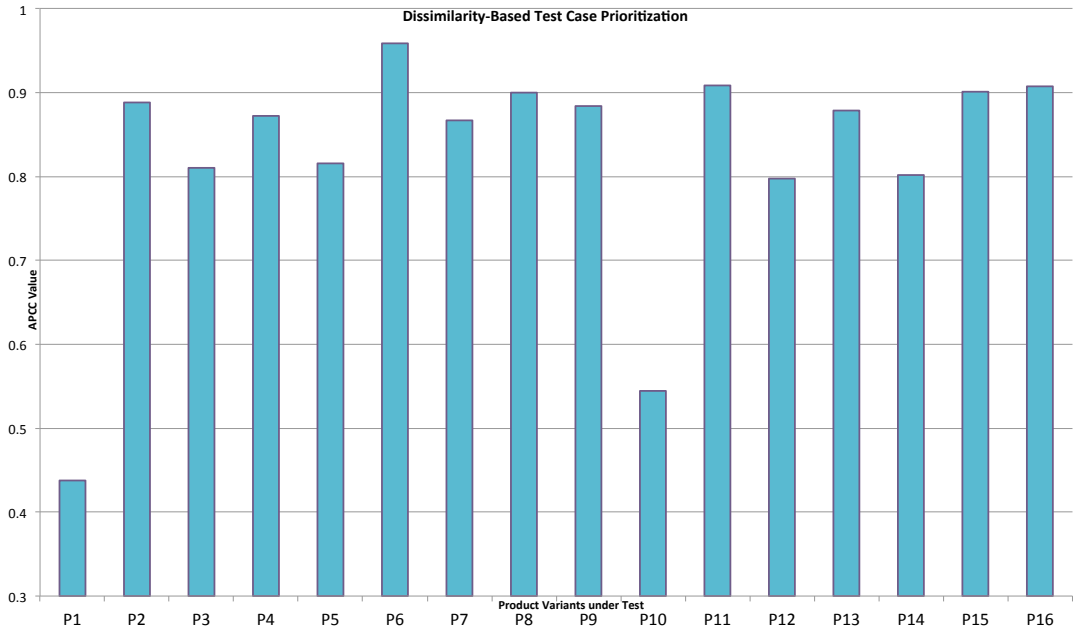

Figure A.4: APCC for BCS with Dissimilarity-Based Test Case Prioritization

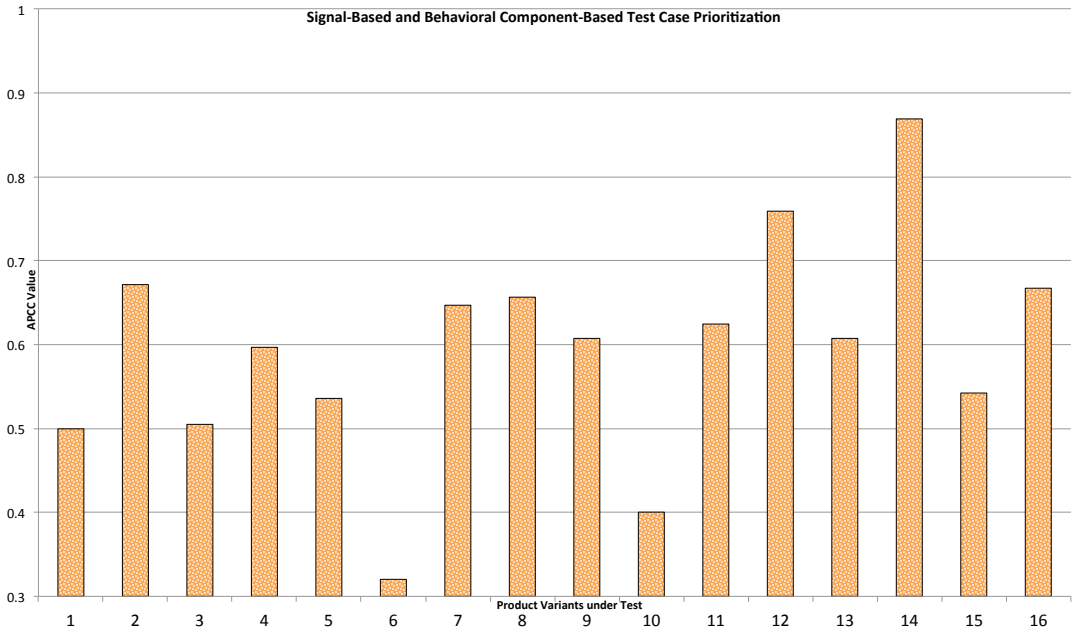

Figure A.5: APCC for BCS with Signal-and Behavioral Component-Based Test Case Prioritization

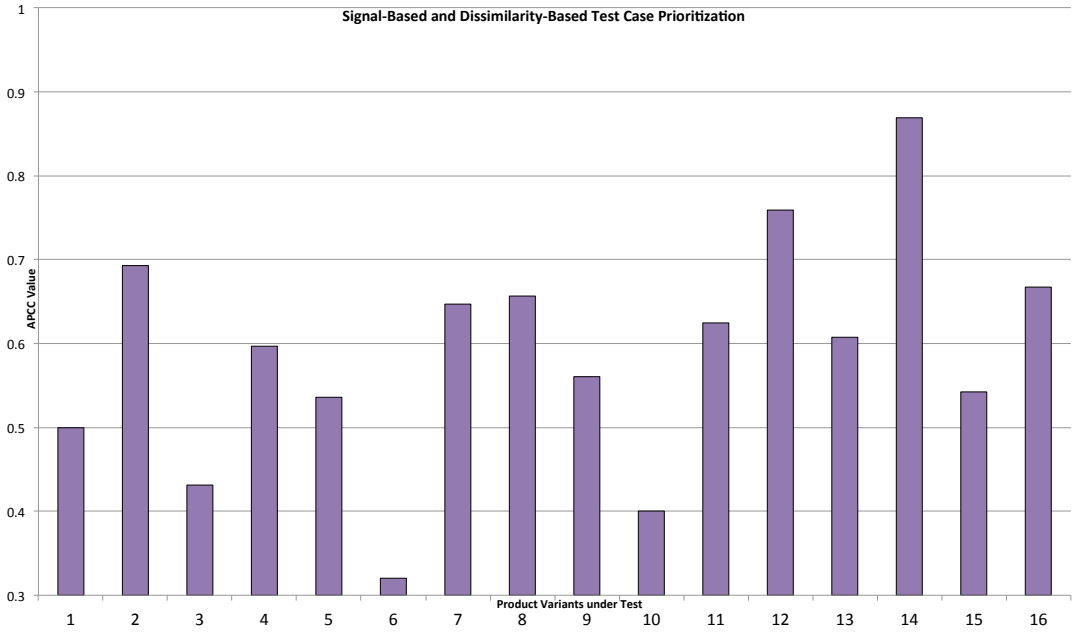

Figure A.6: APCC for BCS with Signal- and Dissimilarity-Based Test Case Prioritization

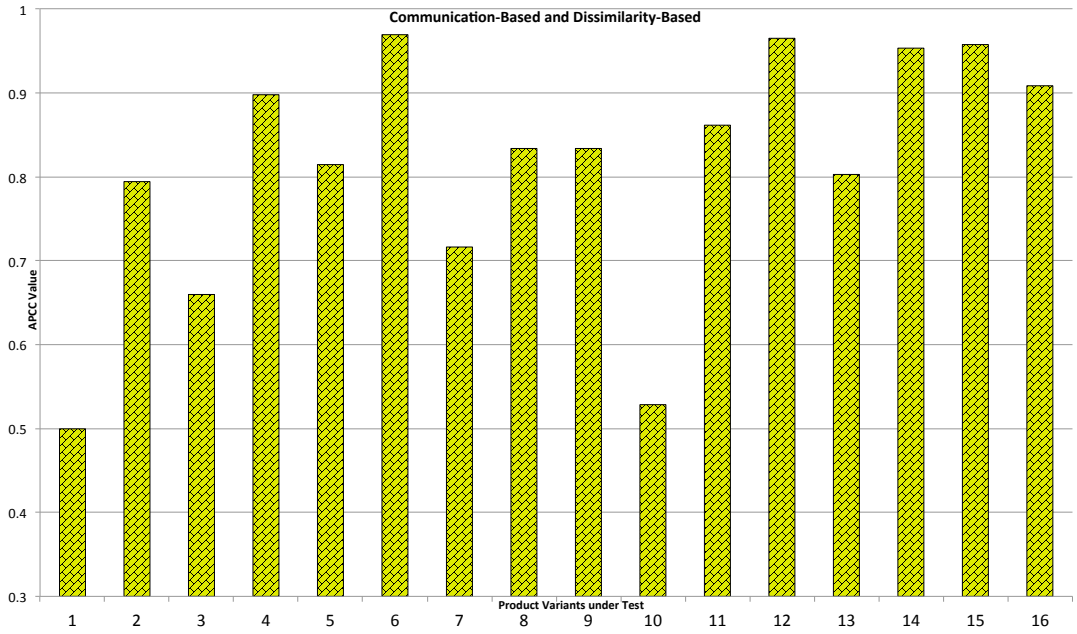

Figure A.7: APCC for BCS with Communication and Dissimilarity-Based Test Case Prioritization

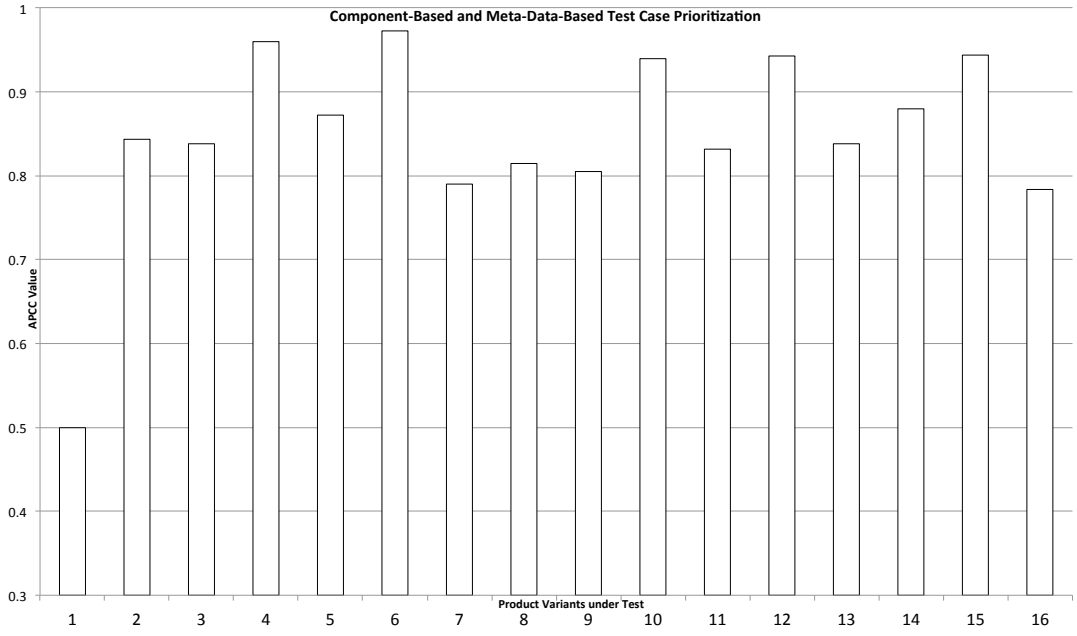

Figure A.8: APCC for BCS with Component- and Meta-Data-Based Test Case Prioritization

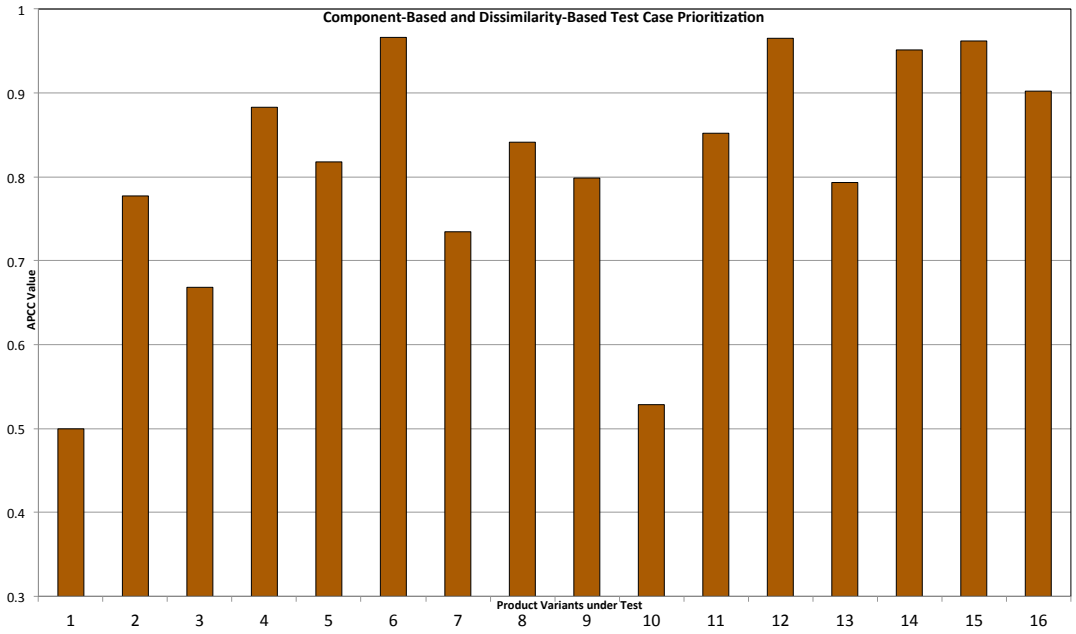

Figure A.9: APCC for BCS with Component and Dissimilarity-Based Test Case Prioritization

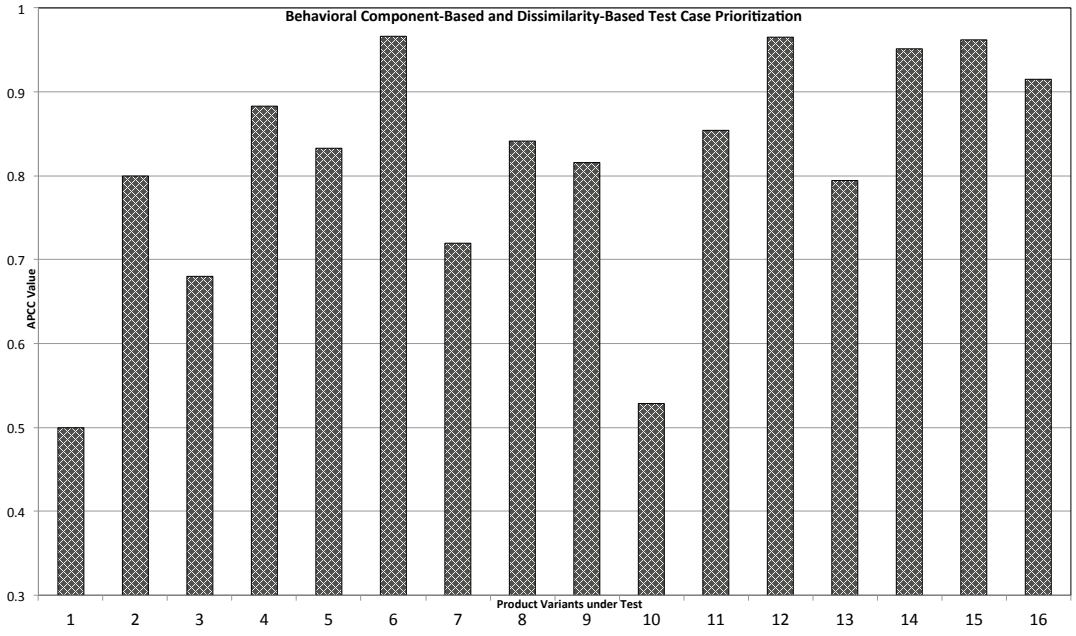

Figure A.10: APCC for BCS with Behavioral Component- and Dissimilarity-Based Test Case Prioritization

B Evaluation of Test Case Selection

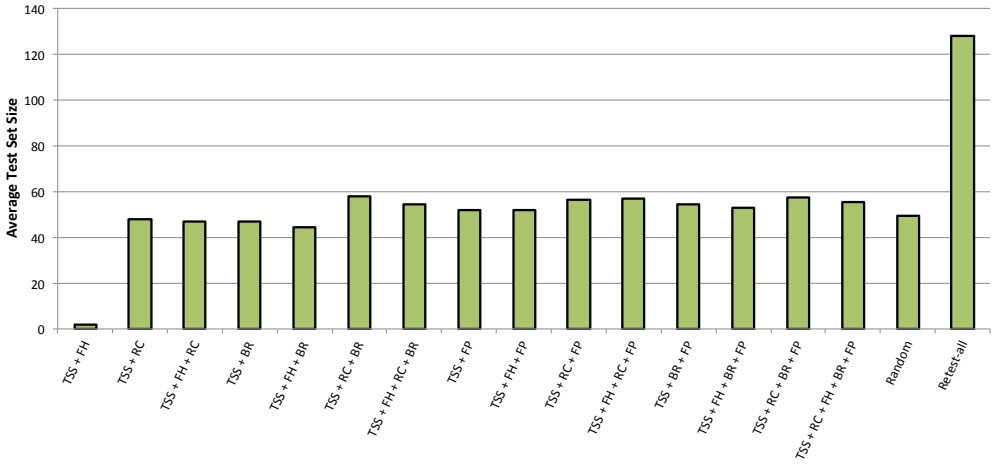

Figure B.1: Average Test Set Sizes for BCS

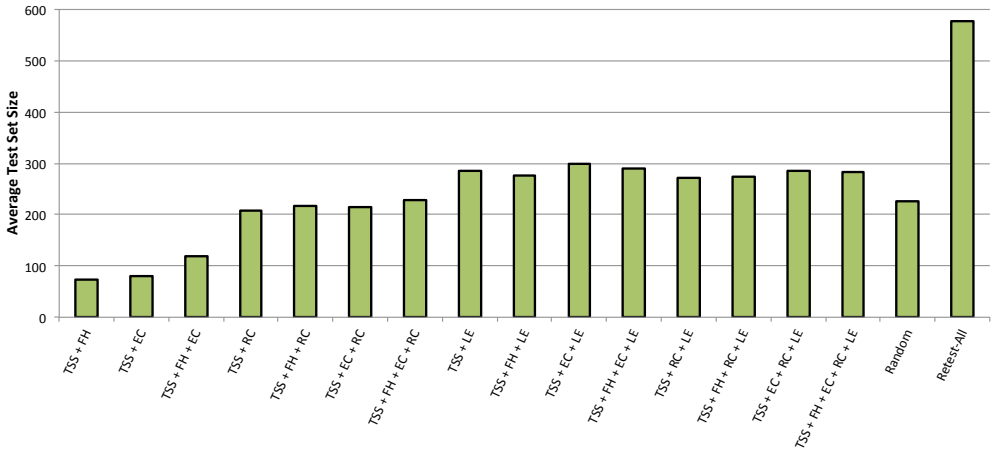

Figure B.2: Average Test Set Sizes for Industry Data

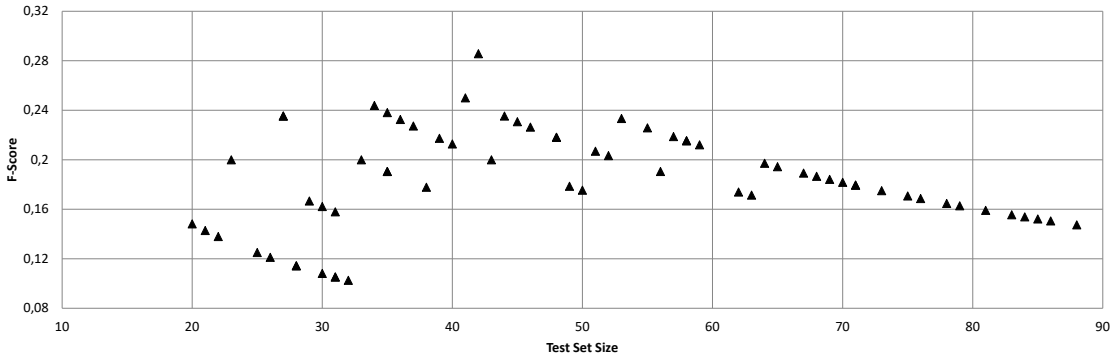

Figure B.3: Best Pareto Front for BCS, obtained by $\{TSS, BR\}$

Table B.1: Mean, Median, Variance and Standard Deviation (SD) of Objective Combinations for both Subject Systems

Objective Combination	Mean	Median	Variance	SD
BCS				
TSS + FH	0.0000	0.0000	0.0000	0.0000
TSS + RC	0.1284	0.1250	0.0018	0.0422
TSS + FH + RC	0.0432	0.0435	0.0008	0.0287
TSS + BR	0.1997	0.1944	0.0025	0.0501
TSS + FH + BR	0.1197	0.1194	0.0018	0.0423
TSS + RC + BR	0.1678	0.1611	0.0017	0.0408
TSS + FH + RC + BR	0.0761	0.0750	0.0013	0.0356
TSS + FP	0.1134	0.1071	0.0011	0.0325
TSS + FH + FP	0.0193	0.0227	0.0004	0.0203
TSS + RC + FP	0.1220	0.1164	0.0010	0.0322
TSS + FH + RC + FP	0.0324	0.0342	0.0006	0.0250
TSS + BR + FP	0.1501	0.1449	0.0018	0.0427
TSS + FH + BR + FP	0.0492	0.0494	0.0012	0.0340
TSS + RC + BR + FP	0.1318	0.1277	0.0016	0.0402
TSS + RC + FH + BR + FP	0.0619	0.0632	0.0012	0.0345
Industry Data				
TSS + FH	0.0275	0.0234	0.0004	0.0209
TSS + EC	0.0470	0.0404	0.0007	0.0274
TSS + FH + EC	0.0112	0.0000	0.0002	0.0139
TSS + RC	0.0981	0.0979	0.0004	0.0201
TSS + FH + RC	0.0258	0.0246	0.0002	0.0129
TSS + EC + RC	0.0736	0.0744	0.0004	0.0196
TSS + FH + EC + RC	0.0174	0.0162	0.0001	0.0112
TSS + LE	0.0737	0.0736	0.0004	0.0189
TSS + FH + LE	0.0356	0.0350	0.0002	0.0132
TSS + EC + LE	0.0751	0.0745	0.0003	0.0182
TSS + FH + EC + LE	0.0334	0.0336	0.0002	0.0128
TSS + RC +LE	0.0745	0.0756	0.0003	0.0172
TSS + FH + RC + LE	0.0395	0.0393	0.0001	0.0121
TSS + EC + RC + LE	0.0813	0.0824	0.0002	0.0152
TSS + FH + EC + RC + LE	0.0376	0.0379	0.0001	0.0118

Index

List of Abbreviations

APCC	= Average Percentage of Changed Covered
APFD	= Average Percentage of Faults Detected
ANN	= Artificial Neural Network
BCS	= Body Comfort System
CB	= Component-based Prioritization
ComB	= Communication-based Prioritization
DSL	= Domain-Specific Language
EC	= Execution Costs
ECU	= Electronic Control Unit
FA	= Failure Age
FC	= Failure Count
FH	= Failure History
FP	= Failure Priority
GA	= Genetic Algorithm
HMI	= Human Machine Interface
LOC	= Lines of Code
MBT	= Model-based Testing
MD	= Meta-Data
ML	= Machine Learning
MPD	= Multi Product Delta
MSC	= Message Sequence Chart
PUT	= Product Variant under Test
QFM	= Quantified Feature Model
RBT	= Risk-Based Testing
RC	= Requirements Coverage
TC	= Test Case
TCD	= Test Case Description
SBST	= Search-based Software Testing
SD	= Standard Deviation
SPL	= Software Product Line
SUT	= System Under Test
SVM	= Support Vector Machine
SVM Rank	= Ranked Support Vector Machine (as defined by Joachims [Joa02])
TSS	= Minimize Test Set Size Objective